CRAY RESEARCH, INC.

CRAY X-MP AND CRAY-1® COMPUTER SYSTEMS

FORTRAN (CFT)
REFERENCE MANUAL

SR-0009

RECORD OF REVISION CRAY RESEARCH, INC. PUBLICATION NUMBER SR-0009

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

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

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

Revision	Description
	July, 1976 - Preliminary distribution, Xerox 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.

PREFACE

The Cray FORTRAN Compiler (CFT) translates FORTRAN language statements into Cray Assembly Language (CAL) programs that make effective use of the CRAY-1 and CRAY X-MP Computer Systems. This manual describes the Cray FORTRAN language in its entirety and related Cray Operating System (COS) characteristics.

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.

This publication is divided into three parts as described below.

PART 1 - CFT DESCRIPTION

Sections 1, 2, and 3 describe the CFT language. The presentation progresses from the identities and forms of basic syntactic elements through the methods of data representation and the evaluation of expressions.

Section 4 describes functions and subroutines.

PART 2 - CFT STATEMENTS

Part 2 presents the CFT statements that conform to the ANSI standards and CFT statements that are an extension of those standards. CFT extensions give the programmer a broader range of capabilities. Extensions that can be conveniently replaced with standard statements are described in Appendix E.

PART 3 - THE CFT COMPILER

Part 3 presents the CFT control statement, directives that control compilation, and techniques for improving the performance of CFT programs.

CONTENTS

PREFACE v

PART ONE - CFT DESCRIPTION

1. CRAY FORTRAN LANGUAGE 1-1

THE CRAY FORTRAN COMPILER 1-1
CONFORMANCE WITH THE ANSI STANDARD 1-1
CONVENTIONS 1-2
ELEMENTS OF THE CFT LANGUAGE 1-3
Character sets 1-3
FORTRAN character set 1-3
Auxiliary character set 1-4
Uppercase/lowercase conversion 1-5
Sequences 1-5
Syntactic items 1-5
Constants 1-5
Symbolic names 1-6
Statement labels 1-6
Keywords 1-7
Operators 1-7
Lists and list items 1-7
FORTRAN statements 1-8
Lines 1-8
Comment lines 1-9
Initial lines 1-9
Continuation lines 1-9
Terminal lines 1-10
Compiler directive lines 1-10
THE EXECUTABLE PROGRAM 1-10
Program units 1-10
The main program 1-12
The subprogram 1-12
Normal execution sequence 1-12
Order of statements and lines 1-13

2. DATA REPRESENTATION AND STORAGE 2-1

TYPES OF DATA 2-1
Data type of an array element 2-2

TYPES OF DATA (continued)
Data type of a function 2-2
CONSTANTS 2-2
Integer constants 2-3
Real constants 2-4
Basic real constant 2-5
Constant followed by a real exponent 2-5
Nonzero real constant range 2-5
Double-precision constants 2-5
Constant followed by a double-precision exponent . . 2-6
Nonzero double-precision constant range 2-6
Complex constants 2-6
Nonzero complex constant range 2-7
Logical constants 2-7
Boolean (octal or hexadecimal) constants 2-8
Character constants 2-8.1
VARIABLES 2-9
ARRAYS 2-10
Array declarators 2-10
Format of an array declarator 2-10
Kinds of array declarators 2-11
Actual array declarators 2-11
Dummy array declarators 2-11
Size of an array 2-12
Array element names 2-12
Array storage sequence 2-12.1
Array element order 2-13
Subscript values 2-15
Dummy and actual arrays 2-15
Adjustable arrays and adjustable dimensions 2-16
Use of array names 2-17
CHARACTER SUBSTRINGS 2-18
STORAGE AND ASSOCIATION 2-18
Storage sequences 2-19
Association of entities 2-19
Definition 2-21
Defined entities 2-21
Undefined entities 2-22
SYMBOLIC NAMES 2-23
Scope of symbolic names 2-24
Global entities 2-24
Local entities 2-24
Classes of symbolic names 2-25
Common blocks 2-25
External functions 2-25
Subroutines 2-26
The main program 2-26
Block data subprograms 2-26
Arrays 2-26
Variables 2-27
Constants 2-27

Classes of symbolic names (continued)
Statement functions 2-28
Intrinsic functions 2-28
Dummy procedures 2-28
NAMELIST group name 2-29

3. EXPRESSIONS 3-1

ARITHMETIC EXPRESSIONS 3-1
Arithmetic operators 3-2
Interpretation of arithmetic operators in expressions 3-2
Precedence of arithmetic operators 3-3
Arithmetic operands 3-3
Primaries 3-4
Factors 3-4
Terms 3-5
Arithmetic expressions 3-5
Data type of arithmetic expressions 3-6
Integer quotients 3-8
Type conversion 3-9
Type integer 3-9
Type real 3-9
Type double-precision 3-9
Type complex 3-10
Type Boolean 3-10
Evaluation of arithmetic expressions 3-10
CHARACTER EXPRESSIONS 3-11
Character expression evaluation 3-11
RELATIONAL EXPRESSIONS 3-12
Arithmetic relational expressions 3-12
Character relational expressions 3-13
LOGICAL EXPRESSIONS 3-14
Logical operators 3-14
Form and interpretation of logical expressions 3-14
Values of logical factors, terms, disjuncts, and expressions 3-16
BOOLEAN (MASKING) EXPRESSIONS 3-17
PRECEDENCE OF ALL OPERATORS 3-18
EVALUATION OF EXPRESSIONS 3-19
Order of evaluation of functions 3-19
PARENTHESES AND EXPRESSIONS 3-20
SUMMARY OF RULES OF INTERPRETATION 3-20

4. SUBROUTINE, FUNCTION, AND SPECIFICATION SUBPROGRAMS 4-1

SPECIFICATION SUBPROGRAMS 4-1
Named common blocks 4-1
PROCEDURE SUBPROGRAMS 4-2
Subroutine subprograms 4-2

PROCEDURE SUBPROGRAMS (continued)
Actual arguments 4-2
Subroutine subprogram restrictions 4-3
Function subprograms 4-3
Statement functions 4-3
Referencing statement functions 4-4
Statement function restriction 4-4
External functions 4-5
Referencing external functions 4-5
Execution of external function references . . . 4-5
Actual arguments for external functions 4-5
Intrinsic functions 4-6
Referencing intrinsic functions 4-6
Intrinsic function restrictions 4-6
Utility procedures 4-6
Function subprogram restrictions 4-7
Execution of function references 4-8
Referencing functions 4-8
Non-FORTRAN subprograms 4-9
ARGUMENTS 4-9
Dummy arguments 4-9
Actual arguments 4-10
Association of dummy and actual arguments 4-10
Variables as dummy arguments 4-11
Arrays as dummy arguments 4-12
Procedures as dummy arguments 4-12
Restrictions on the association of entities 4-13
COMMON BLOCKS 4-14
Extended memory common blocks 4-15
Task common blocks 4-15

FIGURES

1-1 Executable program 1-11
2-1 Array storage sequence 2-13
2-2 Array element arrangement and reference 2-14

TABLES

1-1 Special characters 1-4
1-2 Required order of lines and statements 1-14
2-1 Constant value representation 2-3
2-2 Logical constant representation 2-7
2-3 Subscript evaluation 2-16
3-1 Arithmetic operators 3-2
3-2 Interpretation of operators in expressions 3-2
3-3 Precedence of arithmetic operators 3-3
3-4 Arithmetic operand, expression, and result typing relationships 3-7

TABLES (continued)

3-5 Type conversion in assignment statements 3-8
3-6 Relational operators 3-12
3-7 Logical operators 3-14
3-8 Precedence among all operators 3-18

PART TWO - CFT STATEMENTS

1. FORTRAN STATEMENTS 1-1

2. DATA SPECIFICATION 2-1

DECLARATION AND INITIALIZATION 2-1
PARAMETER statement 2-1
DIMENSION statement 2-2
POINTER statement (CFT extension) 2-3
DATA statement 2-4
DATA statement restrictions 2-6
TYPE STATEMENTS 2-7
INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL type statements 2-7
CHARACTER type statement 2-8
IMPLICIT statement 2-9
IMPLICIT NONE statement (CFT extension) 2-10
ASSOCIATION STATEMENTS 2-11
EQUIVALENCE statement 2-11
Equivalence association 2-12
Array names and array element names 2-12
Restrictions on EQUIVALENCE statements 2-12
COMMON statement 2-13
Common block storage sequence 2-14
Size of a common block 2-14
Common association 2-15
Differences between named commmon and blank common . 2-15
Restrictions on COMMON and EQUIVALENCE statements . . 2-15
INTRINSIC STATEMENT 2-15
SAVE STATEMENT 2-16

3. ASSIGNMENT STATEMENTS 3-1

ARITHMETIC ASSIGNMENT STATEMENT 3-1
LOGICAL ASSIGNMENT STATEMENT 3-2
CHARACTER ASSIGNMENT STATEMENT 3-2
ASSIGN STATEMENT 3-3

4. PROGRAM CONTROL STATEMENTS 4-1

UNCONDITIONAL GO TO STATEMENT 4-1
COMPUTED GO TO STATEMENT 4-2
ASSIGNED GO TO STATEMENT 4-3
ARITHMETIC IF STATEMENT 4-4
LOGICAL IF STATEMENT 4-4
CONDITIONAL BLOCK STATEMENTS 4-5
IF-block 4-5
Block IF statement 4-6
END IF statement 4-6
ELSE IF-block 4-6
ELSE IF statement 4-7
ELSE-block 4-7
ELSE statement 4-7
Conditional block statement execution 4-8
DO STATEMENT 4-8
Terminal statement 4-10
DO variable 4-10
Range of a DO-loop 4-10
Active and inactive DO-loops 4-11
Executing a DO statement 4-11
Loop control processing 4-13
Execution of the range 4-13
Terminal statement execution 4-13
Incrementation processing 4-13
Transfer into the range of a DO-loop 4-14
CONTINUE STATEMENT 4-14
STOP STATEMENT 4-15
PAUSE STATEMENT 4-15
END STATEMENT 4-16

5. INPUT/OUTPUT STATEMENTS 5-1

INPUT/OUTPUT RECORDS 5-1
Formatted records 5-1
Unformatted records 5-2
End-of-file (ENDFILE) records 5-2
End-of-data records 5-2
INPUT/OUTPUT FILES 5-2
RECORD AND FILE POSITIONS 5-3
DATASETS 5-3
INTERNAL RECORDS AND FILES 5-4
SEQUENTIAL ACCESS OPERATIONS 5-4
DIRECT ACCESS OPERATIONS 5-5
Dataset position before data transfer 5-5
Sequential access 5-5
Direct access 5-5
UNITS 5-6

IDENTIFIERS 5-6
Unit identifiers 5-6
Dataset identifiers 5-7
Format identifiers 5-7
READ, WRITE, AND PRINT STATEMENTS 5-8
Control information lists 5-9
Input/output lists 5-10
Input list items 5-10
Output list items 5-11
Implied-DO lists 5-11
DATA TRANSFER 5-12
Direction of data transfer 5-12
Identifying a unit 5-13
Establishing a format 5-13
Data transfer 5-13
Unformatted data transfer 5-14
Formatted data transfer 5-14
Error and end-of-file conditions 5-15
BACKSPACE, ENDFILE, AND REWIND STATEMENTS 5-16
BACKSPACE statement 5-17
ENDFILE statement 5-17
REWIND statement 5-17
INQUIRE STATEMENTS 5-17
Inquiry by dataset name 5-18
Inquiry by unit 5-18
INQUIRE statement restrictions 5-20
OPEN STATEMENT 5-20
CLOSE STATEMENT 5-21
NAMELIST STATEMENT (CFT EXTENSION) 5-23
NAMELIST input 5-24
NAMELIST input variables 5-25
NAMELIST input processing 5-26
User control subroutines 5-26
NAMELIST output 5-28
User control subroutines 5-28
BUFFER IN AND BUFFER OUT STATEMENTS (CFT EXTENSIONS) 5-29
The UNIT function 5-33
The LENGTH function 5-34
RESTRICTIONS ON INPUT/OUTPUT STATEMENTS 5-34
I/O ERROR RECOVERY 5-34

6. FORMAT SPECIFICATION 6-1

FORMAT STATEMENTS 6-1
Format of a format specification 6-2
EDIT DESCRIPTORS 6-3
Interaction between I/O lists and format specifications . 6-6
Positioning by format control 6-7
Internal representation 6-7
Apostrophe and quotation mark editing 6-7

H editing 6-8
Positional editing (T, TL, TR, and X) 6-8
T, TL, and TR editing 6-9
X editing 6-9
Slash editing 6-10
Colon editing 6-10
Dollar sign editing (CFT extension) 6-10.1
P editing 6-11
Numeric editing (BN, BZ, S, SP, SS, I, F, E, D, and G) . . 6-12
BN and BZ editing 6-13
S, SP, and SS editing 6-13
Integer editing 6-13
F editing 6-14
E editing 6-16
D (double-precision) editing 6-17
G editing 6-17
Complex editing 6-19
O (octal) editing (CFT extension) 6-19
Z (hexadecimal) editing (CFT extension) 6-20
L (logical) editing 6-20
A (alphanumeric) editing 6-21
R (right-justified) editing (CFT extension) 6-23
LIST-DIRECTED I/O 6-24
List-directed input 6-24
List-directed output 6-26

7. PROGRAM UNIT SPECIFICATION 7-1

PROGRAM STATEMENT 7-1
FUNCTION SUBPROGRAMS 7-2
Function reference 7-2
Function statement 7-2
Statement function definition statement 7-3
SUBROUTINE AND CALL STATEMENTS 7-5
Subroutine reference 7-5
Execution of a CALL statement 7-5
SUBROUTINE statement 7-6
RETURN STATEMENT 7-6
Execution of a RETURN statement 7-7
Alternate return 7-7
ENTRY STATEMENT 7-8
Referencing a procedure subprogram entry 7-9
Entry association in function subprograms 7-9
ENTRY statement restrictions 7-9
EXTERNAL STATEMENTS 7-10
BLOCK DATA STATEMENTS 7-11

FIGURE

4-1 IF-levels and blocks 4-9

TABLES

5-1 Print control characters 5-15
5-2 Inquiry specifiers and their meanings 5-19
5-3 OPEN specifiers and their meanings 5-22
5-4 CLOSE specifiers and their meanings 5-23
6-1 Edit descriptors with data types 6-5
6-2 Edit descriptors and data types when SEGLDR and the EQUIV directive are used 6-5

PART THREE - THE CFT COMPILER

1. CFT COMPILER I/O 1-1

THE CFT CONTROL STATEMENT 1-1
ERROR MESSAGES DURING PROGRAM EXECUTION 1-11
INPUT TO CFT 1-11
OUTPUT FROM CFT 1-14
Listable output 1-14
Page header lines 1-14
Source statement listings 1-15
BLOCK BEGINS messages 1-15
Table of statement numbers 1-15
Table of names encountered 1-16
Address field 1-16
Name field 1-16
Type field 1-16
Main usage field 1-17
Block field 1-17
Table of parameters encountered 1-18
Table of block names and lengths in octal 1-18
Table of external names 1-19
Table of loops encountered 1-19
Cross-reference information 1-19
Messages 1-20
Program Unit Page Table 1-20
Compiler options 1-20
Using compiler directive lines 1-21
COMPILER DIRECTIVES 1-21
Listable output control directives 1-22
EJECT directive 1-22
LIST directive 1-22

NOLIST directive 1-23
CODE directive 1-23
NOCODE directive 1-23
Vectorization control directives 1-24
VECTOR directive 1-24
NOVECTOR directive 1-25
NORECURRENCE directive 1-26
IVDEP directive 1-27
IVDMO directive 1-27
VFUNCTION directive 1-27
NEXTSCALAR directive 1-28
SHORTLOOP directive 1-29
Integer control directives (INT24, INT64) 1-29
Multiply/divide directives (FASTMD, SLOWMD) 1-30
Flow trace directives (FLOW/NOFLOW) 1-30
Flow trace enable/disable 1-31
FLODUMP utility 1-31
Options 1-32
SETPLIMQ 1-32
ARGPLIMQ 1-32
FLOWLIM 1-33
Scheduler directives 1-33
UNSAFEIF/SAFEIF directives 1-33
BL/NOBL directives 1-33
Dynamic common block directive (DYNAMIC) 1-34
Array bounds checking directive (BOUNDS) 1-34
BOUNDS options 1-34
Optimization directives 1-35
BLOCK directive 1-36
NO SIDE EFFECTS directive 1-36
ALIGN directive 1-37
NOIFCON directive 1-37
RESUMEIFCON directive 1-38
RESUMEDOREP directive 1-38
NODOREP directive 1-38
CVL directive 1-39
NOCVL directive 1-39
Debugging directives (DEBUG, NODEBUG) 1-39
ROLL/UNROLL directives 1-39
EXTERNAL ROUTINES 1-40

2. CRAY FORTRAN PROGRAMMING 2-1

VECTORIZABLE DO-LOOPS 2-1
Qualifications for vectorization 2-1
Entity categories 2-2.1
Dependencies 2-4
Conditional vector loops 2-10

Vectorization with arrays 2-10
Using optimized routines 2-11
Use of optimized routines by CFT 2-12
Conditional statements 2-13
Compressed index references 2-16
General guidelines for vectorization 2-16
BIDIRECTIONAL MEMORY 2-17

TABLES

1-1 Effect of ALLOC, SAVEALL, and BTREG on variable allocation . . 1-10
1-2 Compiler options 1-12
1-3 External routines 1-40
2-1 Array A elements in vector and scalar modes 2-4.1
2-2 Dependency information combinations 2-5

APPENDIX SECTION

A. CHARACTER SET A-1

B. CRAY FORTRAN INTRINSIC FUNCTIONS B-1

C. CRAY FORTRAN UTILITY PROCEDURES C-1

D. CFT MESSAGES D-1

COMPILE-TIME MESSAGES D-2
LOGFILE MESSAGES D-28
INFORMATIVE DEPENDENCY MESSAGES D-30

E. OUTMODED FEATURES E-1

HOLLERITH CONSTANTS E-2
HOLLERITH EXPRESSIONS E-4
Hollerith relational expressions E-5
HOLLERITH FORMAT SPECIFICATION E-6
TWO-BRANCH ARITHMETIC IF STATEMENTS E-6
INDIRECT LOGICAL IF STATEMENTS E-6
FORMATTED DATA ASSIGNMENT E-7
Encode and decode statements E-7
The ENCODE statement E-8
The DECODE statement E-9
EDIT DESCRIPTORS E-9
DOUBLE DECLARATION STATEMENTS E-10
DATA STATEMENT FEATURES E-10
PUNCH STATEMENT E-11
TYPE STATEMENT DATA LENGTH E-11

RANDOM INPUT/OUTPUT OPERATIONS E-13
Creating a dataset for random access E-13
Dataset connection E-13
Positioning while connected for random access (GETPOS/SETPOS) E-13
READMS/WRITMS routines E-14
Modifying a record under random access E-15
Extended range of a DO-loop E-15
Noncharacter arrays for format specification E-15
EOF, IEOF, and IOSTAT functions E-16

F. CREATING NON-FORTRAN PROCEDURES F-1

G. SYMBOLIC DEBUG PACKAGE G-1

H. UNBLOCKED DATASETS H-1

I. REPRIEVE PROCESSING I-1

REPRIEVE INITIATION I-1
REPRIEVE TERMINATION I-2

J. FTREF UTILITY J-1

TABLES

B-1 Generic and specific intrinsic function names B-7
C-1 CFT utility procedures C-2
E-1 Data length E-12

INDEX

PART 1

CFT DESCRIPTION

CRAY FORTRAN LANGUAGE 1

FORTRAN is a system of notation devised for easy and accurate computer program specification. It is a computer programming language that is especially useful for solving mathematical problems. Ordered sets of alphabetic, numeric, and special characters are used to construct FORTRAN statements which, in turn, are ordered to comprise a computer program. FORTRAN permits computer program specification with little dependence upon the characteristics of the computer system to be used.

The Cray FORTRAN (CFT) language is a high-level language that employs the features of the CRAY-1 and CRAY X-MP Computer Systems. The language is an extended version of the American National Standards Institute (ANSI) 77 programming language, FORTRAN, ANSI X3.9-1978, which is often called FORTRAN 77. The CFT Compiler transforms CFT language statements into machine-language instruction sequences or programs. The Cray Operating System (COS) supports CFT and the programs CFT creates by initiating and monitoring their execution.

The CFT language is described in this section. The fundamentals of its notation and syntax are introduced, followed by a description of the elements of an executable program and the CFT method of data representation.

THE CRAY FORTRAN COMPILER

The CFT compiler converts statements in the FORTRAN language to the binary machine language of the Cray Computer Systems. During this conversion, CFT constructs machine-language instruction sequences that apply the full range of the Cray Computer Systems features and capabilities during program execution.

CONFORMANCE WITH THE ANSI STANDARD

Specifications for the Cray FORTRAN language are based on standards established in 1977 by the American National Standards Institute and documented in the publication ANSI FORTRAN X3.9-1978.

Extensions to these standards afford the CFT language programmer a broader range of capabilities. Statement extensions that deviate from the ANSI standards are identified as CFT extensions (See part 2). Other deviations from the ANSI standards are flagged with notes throughout the manual.

CONVENTIONS

The conventions used in this publication to describe the syntax of FORTRAN statement forms consist of ordered sequences of the following elements:

UPPERCASE	Uppercase letters, numbers, and symbols indicate their actual use
Italics	Identify a user-provided item and are also used when terminology is being defined
[] Brackets	Enclose items for optional use
{} Braces	Enclose two or more parameters when only one of the parameters must be used
... Ellipsis	Indicate optional use of the preceding item one or more times in succession

Except where specifically stated that blanks are required, blank characters are not needed and only enhance readability.

Example:

PRINT *f*[,*iolist*]

where *f* is a FORMAT statement identifier, and

iolist is an I/O list

describes the syntactical construct beginning with the letters PRINT, followed by those symbols identifying a FORMAT statement identifier and, optionally, a comma and one or more sets of symbols identifying I/O list items separated by commas.

The FORTRAN language statements

```
PRINT 88

PRINT 1234,A,B,C,X,Y,Z

PRINT6,VALUE

PRINT 0054,ALPHA,BETA,GAMMA,DELTA,ETCETERA
```

thus comply with this form, provided that the use of FORMAT statement identifiers and I/O list items is proper.

ELEMENTS OF THE CFT LANGUAGE

The CFT language is composed of numbers and letters and the special characters identified in the character sets below. Certain sequences of these are called syntactic items and can be grouped into FORTRAN statements which, in turn, are ordered into program units.

CHARACTER SETS

Two sets of characters are used in CFT language notation, the FORTRAN character set and the auxiliary character set. Uppercase and lowercase letters, digits, and certain special characters belong to the FORTRAN character set. All other characters representable in the Cray computer systems belong to the auxiliary character set. Appendix A describes these characters and their internal codes.

FORTRAN character set

The *FORTRAN character set* consists of the 26 uppercase *letters*, A-Z; the 26 lowercase *letters*, a-z; the 10 *digits*, 0-9; and the 14 *special characters* described in table 1-1. An *alphanumeric character* is any letter or digit.

The 8-bit ASCII internal code for each of these characters is given in Appendix A. The relative magnitudes of these internal codes establish their *collating sequence*. Digits precede letters in this collating sequence.

The ANSI FORTRAN Standard does not specify a collating sequence except within the letter group (A-Z) and the digit group (0-9).

Table 1-1. Special characters

Symbol	Name
	Blank or space
=	Equals
+	Plus
-	Minus or hyphen
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Dollar sign/currency symbol
'	Apostrophe
"	Quotation mark
:	Colon

The ANSI FORTRAN Standard does not provide for quotation marks.

Auxiliary character set

Appendix A contains the complete Cray Computer Systems set of characters and the codes used for the internal representation of each. Those characters not in the FORTRAN character set are members of the *auxiliary character set* and are nonstandard.

The ANSI FORTRAN Standard does not specify an auxiliary character set.

UPPERCASE/LOWERCASE CONVERSION

A FORTRAN program and its input data, if any, generally use uppercase and lowercase letters interchangeably. CFT lists the source program as it is received. Secondary listings, such as error messages or cross reference lists, have all lowercase letters converted to uppercase. The only exception to this is within character or Hollerith constants, where no case conversion occurs.

The ANSI FORTRAN Standard does not provide for lowercase letters.

SEQUENCES

A *sequence* is a set of n elements ordered in a one-to-one correspondence with the ordinals 1,2, ... ,n. An *empty sequence* contains no elements.

SYNTACTIC ITEMS

Syntactic items of the FORTRAN language are formed with sequences of FORTRAN character set elements. They include the following.

- Constants
- Symbolic names
- Statement labels
- Keywords
- Operators
- Special characters

Within syntactic items, uppercase and lowercase letters can be used interchangeably.

Constants

A *constant* is a syntactic item representing an unvarying value. Several types of constants are illustrated below and are more fully described in part 1, section 2, under the subheading, Constants.

Examples:

Representation	Type	Value
1024	Integer	1024
10.E1	Real	100.
10.e1	Real	100.
1.5	Real	1.5
.FALSE.	Logical	false
.TRUE.	Logical	true
72.	Real	72.
'CRAY-1'	Character	CRAY-1
75.63D-2	Double precision	.7563
(6.1,-3.2)	Complex	$6.1+(-3.2)\sqrt{-1}$

Symbolic names

A *symbolic name* declares or references a program unit, procedure, or value. It is composed of one to eight alphanumeric characters. The first character must be a letter. Leading, trailing, and embedded blank characters are ignored.

The ANSI FORTRAN Standard limits a symbolic name to a maximum of six characters.

Examples:

DATAONE	F293	SIN ALPHA
DATA1	U 238	TEST1234
SAM	Sam	sAM

(The three names on the last line are equivalent).

Statement labels

A *statement label* uniquely identifies a statement in a program unit to permit its being referenced by other statements in the same program unit. A statement label is composed of one to five digits, one of which must be nonzero. Leading zeros and leading, trailing, and embedded blank characters do not alter the identity of a statement label. For example, in the following statement sequence, 22 and 2 2 refer to the statement label 22.

```
      IF (X) 22,11,2 2
22    CONTINUE
```

Keywords

A *keyword* is a prespecified sequence of letters having special significance in FORTRAN language statements. Some examples of keywords are INTEGER, WRITE, and GO TO. Leading, trailing, and embedded blanks occurring in a keyword are ignored. Duplication of a keyword as a symbolic name poses no problem because of the context in which each is used.

Operators

An *operator* specifies arithmetic, relational, logical, and character operations within program units. An operator is expressed as one or two special characters or a combination of special characters and letters. Leading, trailing or embedded blanks do not affect the identity of an operator.

Examples:

Representation	Type	Meaning
+	Arithmetic	Addition
**	Arithmetic	Exponentiation
.AND.	Logical	Intersection
.EQ.	Relational	Equal to
//	Character	Concatenation

LISTS AND LIST ITEMS

A *list* is a sequence of one or more syntactic items separated, if more than one, by the special character comma. The syntactic items appearing in a list are called list items. Blank characters preceding, following, or embedded within *list items* do not affect their interpretation.

Examples:

```
A,B,C,D,E                 701,55,100

ARRAY1, VALUE2,X,ABC      UNO,2,TRES
```

FORTRAN STATEMENTS

A FORTRAN *statement* is a sequence of syntactic items that usually begins with a keyword. As a fundamental component in a FORTRAN program specification, the FORTRAN statement describes either the form of data and program elements or the actions to be taken by the program. A statement label can precede a statement, but is not a part of the statement itself.

The type of a statement is indicated by the keyword it contains or by its form. The total number of characters used to express a statement is limited to 1,320, including blank characters. Aside from this character-count limitation, leading, trailing, and embedded blank characters do not affect statement interpretation.

A statement is classified as either executable or non-executable. An *executable statement* is one that specifies an action. A *non-executable statement* is an inactive descriptor of data or program form. CFT statements appear in part 2.

LINES

A single row of information is a *line*. A line can contain up to 96 columns. Columns 73 through 96 are unused by CFT. (A blank position in the row or an unpunched column of a card represent the special character blank.) All notation required to describe a FORTRAN program is expressed as an ordered sequence of the following types of lines.

- Comment
- Initial
- Continuation
- Terminal
- Compiler directive

The ANSI FORTRAN Standard limits line length to 72 characters.

Comment lines

A *comment line* is a descriptive commentary or a blank line that can have the letter C or an asterisk in column 1, or only blank characters in columns 1 through 72. (See compiler directive lines.) The contents of columns 2 through 96 of a comment line have no effect on the FORTRAN program being created.

Comments can also be embedded in any statement except a FORMAT statement. When an exclamation point (!) appears outside a quoted string, the remainder of the line is treated as a comment. The exclamation point cannot appear in columns 1-5. An exclamation point in column 6 indicates continuation of the previous statement, not continuation of an embedded comment on the previous line.

Examples:

```
10    X=Y*Z              !Compute the product
     !   +SUM            !and add it to the sum
```

The ANSI FORTRAN Standard does not provide for embedded comments.

Initial lines

The *initial line* expresses all or the initial part of a single FORTRAN statement in columns 7 through 72. This line can have a statement label of one to five digits and/or blank characters in columns 1 through 5. An initial line has neither the letter C nor an asterisk in column 1, and must have either the digit 0 or a blank character in column 6. A *terminal line* is a special form of initial line.

Continuation lines

One or more *continuation lines* can be used to extend an initial line when expressing a single FORTRAN statement. A continuation line has neither the letter C nor an asterisk in column 1. It has a character other than 0 or blank in column 6, and contains a portion of a FORTRAN statement in columns 7 through 72. Columns 1 through 5 must contain only blanks. A sequence of one initial line followed by up to 19 continuation lines can be used for a single FORTRAN statement. This sequence of lines can have any number of comment lines interspersed. The initial line of such a sequence must not appear to be a terminal line.

Terminal lines

A single *terminal line* must be used as the last line of every program unit. A terminal line is a special form of initial line that completely contains an END statement (that is, the letters E, N, and D appearing in that order anywhere in columns 7 through 72). This line contains no other characters.

Compiler directive lines

A line having the characters CDIR$ in columns 1 through 5 is a *compiler directive line* and can contain one or more compiler directives. These lines and their compiler directives are described in part 3, section 1.

The ANSI FORTRAN Standard does not provide for compiler directives.

THE EXECUTABLE PROGRAM

An *executable program* is an ordered set of FORTRAN statements grouped into one or more program units. Certain program units can reference pre-established procedures called subprograms. Computer program specifications are, therefore, established from the following two sources.

- The FORTRAN statements comprising the executable program
- Subprograms referenced by these FORTRAN statements

Figure 1-1 illustrates these program units, the procedures they reference, and the overall organization of these entities in the executable program.

PROGRAM UNITS

A *program unit* contains a sequence of statements and optional comment lines. A set of program units in an executable program must include one main program and can also include one or more subprograms.

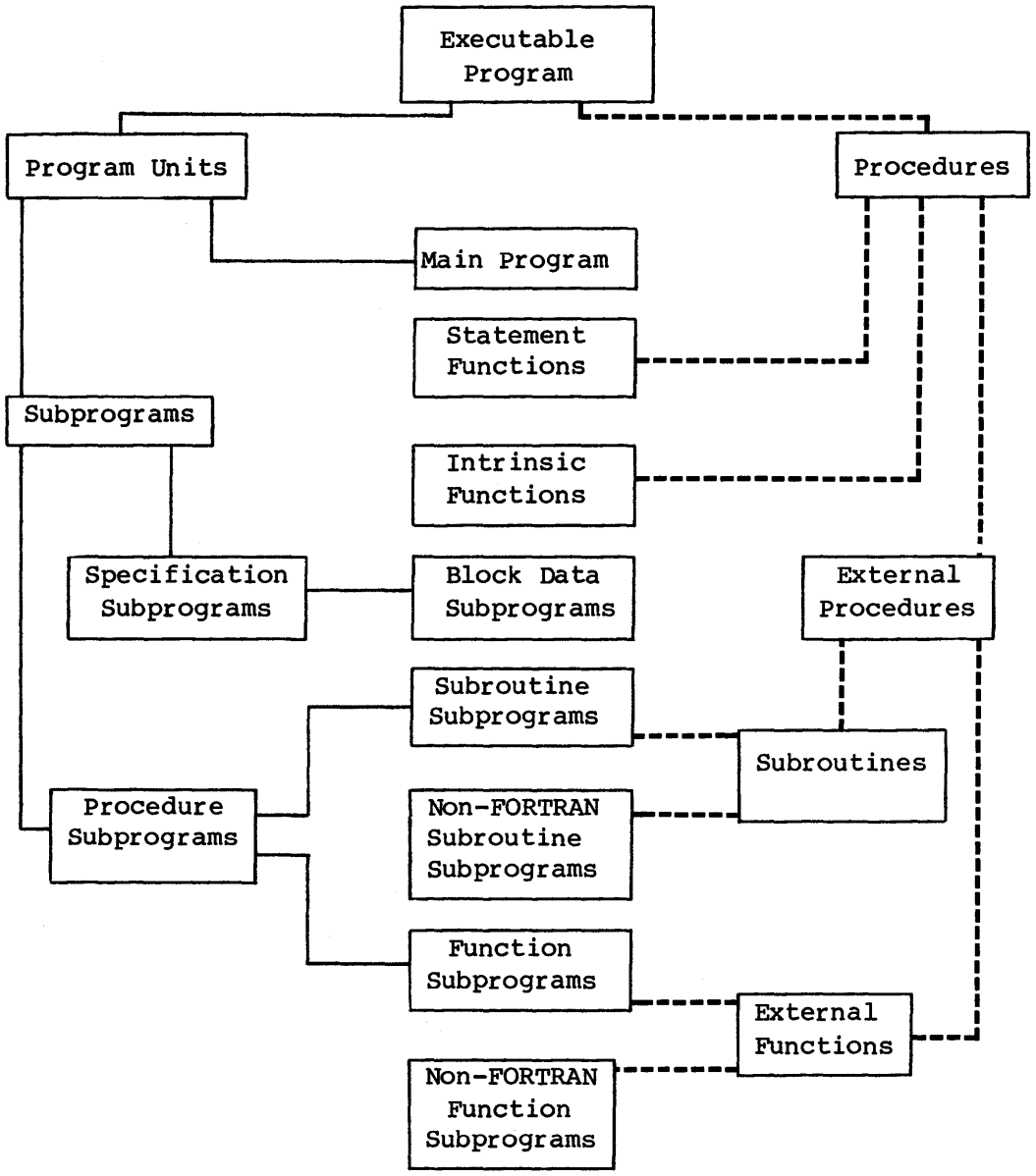


Figure 1-1. Executable program

The main program

A *main program* is a program unit where a FUNCTION, SUBROUTINE, or BLOCK DATA statement does not appear as its first statement. An executable program must contain only one main program. Program execution begins with the first executable statement of the main program.

Optionally, the first statement of a main program can be a PROGRAM statement. Using the PROGRAM statement is strongly recommended since several compiler options, such as F and H, depend on the presence of a PROGRAM statement. The PROGRAM statement must be the first statement of the main program.

The subprogram

A *subprogram* is a program unit having a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. A subprogram must not 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. (See section 4 for a detailed description of subprograms).

NORMAL EXECUTION SEQUENCE

Program execution begins with the first executable statement of the main program. A *normal execution sequence* is the execution of executable statements in their order of appearance in a program unit. When a subprogram is referenced, execution begins with the first executable statement of that subprogram. When a subprogram entry is referenced (see the ENTRY statement in part 2, section 7), execution begins with the first executable statement following the ENTRY statement named in that reference.

The following statements can alter the normal execution sequence.

- Unconditional, assigned, and computed GO TO
- Arithmetic IF
- RETURN
- PAUSE
- CALL with an alternate return specifier
- STOP

- An I/O statement containing an error specifier or an end-of-file specifier
- Logical IF containing the preceding forms
- DO
- Block IF or ELSEIF
- The terminal statement of a DO-loop
- The last statement of an IF-block or ELSE IF-block
- END

Normal execution sequence is not affected by the interspersing of non-executable statements or comment lines among executable statements.

A procedure subprogram must not be referenced twice without the execution of a RETURN or END statement in that procedure.

ORDER OF STATEMENTS AND LINES

The FORTRAN language requires that the various types of statements and lines appear in a specific order. Table 1-2 illustrates the required order of statements and lines for a program unit. Vertical lines delimit varieties of statements that can be interspersed. For example, FORMAT statements can be interspersed with PARAMETER, DATA, executable, and statement function definition statements.

Horizontal lines delimit varieties of statements that must not be interspersed. For example, statement function definition statements must not be interspersed among executable statements.

The top-to-bottom order indicates the first-to-last appearance of lines and statements in a program unit. Note that an END statement must appear in the last line of a program unit and cannot be followed by a comment line intended as a part of that same program unit.

Table 1-2. Required order of lines and statements

<table>
<tr><td rowspan="5">Comment and compiler directive lines*</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement</td></tr>
<tr><td rowspan="4">ENTRY and FORMAT statements</td><td rowspan="2">PARAMETER statements*</td><td>IMPLICIT statements*</td></tr>
<tr><td>Other specification statements</td></tr>
<tr><td rowspan="2">DATA statements</td><td>Statement function definition statements</td></tr>
<tr><td>Other executable statements</td></tr>
<tr><td colspan="4">END statement</td></tr>
</table>

* Note the restrictions described for the interspersing of IMPLICIT and PARAMETER statements (see part 2, section 2) and for compiler directive lines (see part 3, section 1).

DATA REPRESENTATION AND STORAGE 2

Data can be specified in a FORTRAN program as a constant, a variable, an array, or an array element. Data can be created by a function when that function is referenced. A constant, variable, array element, or function *reference* occurs when a symbolic name appears in a context where a value is required. A reference to a variable or array element provides the value currently contained by that entity without modifying that value.

Reference to a constant provides an invariant value, which cannot be modified. Reference to a function causes a value to be defined.

TYPES OF DATA

The data type of a constant, a variable, an array, an array element, or a function can be specified as one of the following.

- Integer - Integral, signed values
- Real - Signed, mixed-value approximations (integral number plus fraction)
- Double-precision - Signed, mixed-value approximations extended to approximately twice the precision of real data
- Complex - Values that approximate complex values as pairs of signed, mixed-value approximations (the first member of a pair for the real part, the second member for the imaginary part)
- Logical - Values that represent the logical values true and false
- Character - Sequences of characters
- Boolean - Octal values representing the binary contents of Cray computer words

---

The ANSI FORTRAN Standard does not provide for Boolean data specification.

Once a symbolic name is identified with a particular type, the type of that name is implied for all usages of that name.

The data type of a 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 explicitly specified, a first letter of I, J, K, L, M, or N implies type integer; any other first letter implies type real. The default implied typing can be changed or confirmed by an IMPLICIT statement. (See the IMPLICIT statement and type statements in part 2, section 2.)

DATA TYPE OF AN ARRAY ELEMENT

The data type of an array element is the same as the data type of the array within which it exists.

DATA TYPE OF A FUNCTION

The data type of a function establishes the type of data provided when the function is referenced in an expression.

The name of an *intrinsic function* is prespecified to agree with the type of data provided. It cannot be explicitly or implicitly retyped. Intrinsic functions are listed in Appendix B.

The data type of a function subprogram is implied by its name. It can be retyped by a type statement or an IMPLICIT statement in the function subprogram. It can also be specified in the FUNCTION statement that names the subprogram.

CONSTANTS

Within an executable program, all constants expressed in the same form have the same invariant value. The value zero is neither positive nor negative. A signed zero has the same value as an unsigned zero.

The form of the character sequence representing a constant specifies both its value and its data type. A PARAMETER statement allows a constant to be given a symbolic name. The data type of the symbolic name of a constant is specified in the subsection, Types of Data, described earlier in this section).

Except within character constants, blank characters occurring in a constant do not effect its value.

Integer, real, double-precision, and complex constants are *arithmetic constants*. Table 2-1 shows examples of values with their integer, real, double-precision, and complex representations.

An *unsigned constant* is an arithmetic constant without a leading sign. A *signed constant* has a leading plus or minus. An *optionally signed constant* can be either signed or unsigned. Arithmetic constants are optionally signed except where otherwise specified.

Table 2-1. Constant value representation

Value	Integer Constant	Real Constant	Double-precision Constant	Complex Constant
0	0	0.	0D	(0.,0.)
692	692	692. 692.0 692E0 692.E0 692.0E0 6920E-1 .692E3 6.92E2	 692D0 692.D0 692.0D0 6920D-1 .692D3 6.92D2	(692.,0.) (692.0,0.) (692E0,0.) (692.E0,0.) (692.0E0,0.) (692.0E-1,0.) (.692E3,0.) (6.92E2,0.)
6.128547472		6.128547472 6.128547472E0 6128547472E-9 6128547472.0E-9 .6128547472E1 612.8547472E-2	 6.128547472D0 6128547472D-9 6128547472.0D-9 .6128547472D1 612.8547472D-2	(6.128547472,0.) (6.128547472E0,0.) (6128547472E-9,0.) (6128547472.0E-9,0.) (.6128547472E1,0.) (612.8547472E-2,0.)
$.875 \cdot \sqrt{-1}$				(0.,.875) (0.,875E-3) (0.,.875E0) (0.,8.75E-1) (0.,.000000875E6)
$692 + .875 \cdot \sqrt{-1}$				(692.,.875) (692E0,0.875) (69.2E1,875E-3) (.692E3,875.E-3) (6.92E2,8.75E-1)

INTEGER CONSTANTS

Integer data represents values that are positive, negative, or zero. An integer data item occupies one storage unit in a storage sequence.

The form of an *integer constant* is an optional sign followed by a non-empty sequence of digits specifying a decimal integer value.

Integer constants are represented in the Cray Computer Systems by integral binary values (I) in the range

$$-2^{63} \leq I < 2^{63}.\dagger$$

This is approximately the decimal range

$$0 \leq |I| < 10^{19}.$$

(Special reduced ranges of integer constants are discussed in part 3, section 1 under Compiler Directives.)

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

NOTE

The negative of a nonzero constant, exponent value, or complex portion is formed by preceding its expression with a minus sign. The use of a plus sign in this position or the absence of either sign denotes a positive constant.

REAL CONSTANTS

Real data is an approximation to the value of a real number, assuming a positive, negative, or zero value. Real data occupies one storage unit in a storage sequence.

A *real constant* can be expressed as one of the following.

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

---

† Use the value -2^{63} with caution since its use with arithmetic and relational operators often causes an undetected overflow.

Basic real constant

A *basic real constant* consists of an optional sign, an integer portion, a decimal point, and a fractional portion, in that order. Both the integer portion and the fractional portion are sequences of digits representing integral and fractional decimal values, respectively. 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 value; the excess digits are lost by CFT in roundoff.

Constant followed by a real exponent

The form of a *real exponent* is the letter E followed by an optionally signed integer constant. The real exponent represents a power of 10. The constant is multiplied by the power of 10. The decimal point in a basic real constant is optional if there is no fractional portion and if a real exponent is specified.

Nonzero real constant range

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

$$2^{-8191} \leq |R| < 2^{8191}$$

Nonzero real constants 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 to the following decimal range with approximately 14 decimal digits of precision.

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

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

DOUBLE-PRECISION CONSTANTS

Double-precision data is an approximation to the value of a real number with approximately twice the precision of real data. Double-precision data can be positive, negative, or zero, and it occupies two consecutive storage units in a storage sequence.

A *double-precision constant* can be expressed as one of the following.

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

Basic real constants and integer constants are defined in previous subsections.

Constant followed by a double-precision exponent

The form of a *double-precision exponent* is the letter D followed by an optionally signed integer constant. The double-precision exponent represents a power of 10. The constant is multiplied by the power of 10. The decimal point in a basic real constant is optional if there is no fractional portion and if a real exponent is specified.

Nonzero double-precision constant range

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

$$2^{-8191} \leq |D| \leq 2^{8191}$$

Nonzero double-precision constants 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 to the following decimal range with approximately 29 decimal digits of precision.

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

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

COMPLEX CONSTANTS

Complex data approximates the value of a complex number and is represented by a pair of real data items. The first member of the pair represents the real portion and the second, the imaginary portion of the data. Complex data occupies two consecutive storage units in a storage sequence: the first for the real portion and the second for the imaginary portion.

The form of a *complex constant* is an ordered pair of optionally signed real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real portion of the complex constant and the second is the imaginary portion.

Nonzero complex constant range

Nonzero complex constant components (where $C=C_{real}+iC_{imag}$) are represented in the Cray Computer Systems by two normalized, floating-point binary values (C_{real},C_{imag}) in the following range.

$$2^{-8191} \leq |C_{real}|, |C_{imag}| \leq 2^{8191}$$

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.

$$10^{-2466} < |C_{real}|, |C_{imag}| < 10^{2466}$$

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

LOGICAL CONSTANTS

Logical data can assume only the logical values true and false. Logical data occupies one storage unit in a storage sequence.

The forms, values, and internal representations of a *logical constant* are shown in table 2-2.

Table 2-2. Logical constant representation

Form	Value	Internal representation
.TRUE. or .T.	true	A negative value
.FALSE. or .F.	false	A zero or positive value

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

BOOLEAN (OCTAL OR HEXADECIMAL) CONSTANTS

Boolean data is a set of binary zeros and ones that accounts for the content of each bit position in a single storage unit (64-bit Cray computer word).

The ANSI FORTRAN Standard does not provide for Boolean constants.

A *Boolean constant* can be represented 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. When all 22 octal digits express a Boolean constant, their binary equivalents correspond with the content of every bit position in the storage unit (64-bit word). In this case, the leftmost octal digit can be a 0 or a 1 only, specifying 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).

The hexadecimal form of a Boolean constant contains the letter X followed by a string of 1 to 16 hexadecimal digits (0-9, A-F) enclosed in apostrophes or quotation marks. The hexadecimal digits may be preceded by an optional + or - sign. Blanks are insignificant in hexadecimal constants. When all 16 hexadecimal digits express a Boolean constant, their binary equivalents correspond with the content of every bit position in the storage unit (64-bit word).

When the Boolean constant contains less than 22 octal digits or 16 hexadecimal digits, the constant is right-justified with zeros filling the leftmost bit positions.

Examples:

Boolean constant	Internal representation (octal)
1274653312572676113745B	1274653312572676113745
0B	0000000000000000000000
17777777777777777777B	17777777777777777777
77740B	0000000000000000077740
00776B	0000000000000000000776
X'ABE'	0000000000000000005276
X"2F0"	0000000000000000001360

Boolean constant	Internal representation (octal)
X"-340"	177777777777777776300
X'1 2 3'	000000000000000000443
X'FFFFFFFFFFFFFFFF'	177777777777777777777

CHARACTER CONSTANTS

A *character constant* consists of any ASCII characters listed in Appendix A as being capable of internal representation.

The form of a character constant is an apostrophe followed by a nonempty string of characters followed by an apostrophe. An optional form of a character constant is a character string delimited by two quotation marks.

The delimiting apostrophes or quotation marks are not part of the data represented by the constant. Two adjacent apostrophes within a string bounded by apostrophes or two adjacent quotation marks within a string bounded by quotation marks are interpreted as a single apostrophe or quotation mark, respectively, and not as a string delimiter. In a character constant, blanks embedded between delimiting apostrophes or quotation marks are significant.

The length of a character constant is the number of characters between its delimiters. However, each pair of consecutive apostrophes or quotation marks counts as a single character. The length of a character constant must be greater than 0 and less than 1317. This limitation is due to the number of lines allowed in a CFT FORTRAN statement.

The ANSI FORTRAN Standard does not provide for the use of quotation marks as delimiters.

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

The ANSI FORTRAN Standard does not specify how character constants are internally represented.

Example:

Character constant	Internal value	Internal representation (octal)
'ABC'	ABC	0405022062004010020040
''''	'	0234401002004010020040
"'"	'	0234401002004010020040
"ABC"	ABC	0405022062004010020040

VARIABLES

A *variable* is an entity that has both a name and a type. Variables can be identified, defined, and referenced.

The type of a variable is optionally specified by the appearance of the variable name in a type statement or an IMPLICIT statement. If the type is not so specified, it is implied by the first letter of the variable name. Variables beginning with the letters I through N are of type integer. All others are of type real.

At any given time during program execution, a variable is either defined or undefined.

ARRAYS

An *array* is a sequence of data that occupies consecutive storage units. Each item in the sequence is an *array element*. Ordered groups of array elements are *array dimensions*. An *array name* identifies an array and the type of data it contains. An *array element name* is an array name suffixed by a subscript that indicates the placement of an element in the array.

The name of an array implicitly identifies all elements of that array as being of the same data type as the array. The name of an array and the names of its elements are local to the program unit where each appears.

ARRAY DECLARATORS

An *array declarator* specifies an array's name, the number of dimensions it contains, and the number of elements in each dimension. The array declarator specifies, therefore, the size of the array and the amount of storage space to be allocated for the array. An array can be specified only once within a given program unit. Array declarators are expressed as list items in certain non-executable FORTRAN statements.

Format of an array declarator

The format of an array declarator is

$$a([d_1:]d_2[,[d_1:]d_2]...)$$

where a is the name of the array and

$[d_1:]d_2$ is a dimension declarator

where d_1 is the lower bound of a dimension declarator and

d_2 is the upper bound of a dimension declarator.

A *dimension declarator* specifies the number of array elements in one dimension of an array. This number is the integer value of the upper bound minus the lower bound of the dimension declarator, plus 1.

The lower and upper bounds of dimension declarators are arithmetic expressions that can contain constants, symbolic names of constants, functions, array elements, or variables. If the type of an entity is real, the entity is truncated to integer. Functions, array elements, and variables can be used only in adjustable array declarators. (See the description of kinds of array declarators, later in this section). If the lower bound is omitted, its value is assumed to be 1. The bound d_1 must be less than or equal to d_2.

The ANSI FORTRAN Standard does not permit dimension declarators to be functions, array elements, or noninteger variables.

The number of dimension declarators specified in an array declarator indicates the number of dimensions in the array. The minimum number of dimensions is one and the maximum is seven.

When using FORMAT as an array name, note that CFT treats FORMAT statements as special cases to allow asterisk edit descriptors. The example

```
    110  FORMAT(I3*I4)=(I5*I6)
```

is ambiguous. A statement is identified by CFT as a FORMAT statement if it has a statement label and begins with the characters FORMAT(. Therefore, an assignment of an array named FORMAT may not have a statement label as in the following example.

```
    110  CONTINUE
         FORMAT(I3*I4)=(I5*I6)
```

Kinds of array declarators

An array declarator is either an actual array declarator or a dummy array declarator.

Actual array declarators - An *actual array declarator* is a constant array declarator having an array name that is not a dummy argument. An actual array declarator is permitted in a DIMENSION, COMMON, or type statement.

Dummy array declarators - A *dummy array declarator* can be a constant or an adjustable or assumed-size array declarator. An *adjustable array declarator* is an array declarator that contains one or more variables. An *assumed-size array declarator* is an array declarator in which the upper bound of the last dimension declarator is an asterisk.

Dummy array declarators appear only in function or subroutine subprograms. They are permitted in DIMENSION or type statements, but not in COMMON statements. An array name used in a dummy array declarator in a subprogram must also appear as an argument in its FUNCTION, SUBROUTINE, or ENTRY statement.

SIZE OF AN ARRAY

The size of an array is the number of elements in the array and is equal to the product of the sizes of all dimensions for that array.

CFT allows a maximum array size of 4,194,304 Cray computer words. The Cray Computer System being used, the memory required for other than the executable program and related data storage, and the executable program size might further restrict the array size.

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

ARRAY ELEMENT NAMES

The format of an array element name is

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

where *a* is the array name,

(s[,s]...) is a subscript, and

s is a subscript expression.

The number of subscript expressions should equal the number of dimension declarators in the array declarator. Fewer subscript expressions cause a warning message to be issued.

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

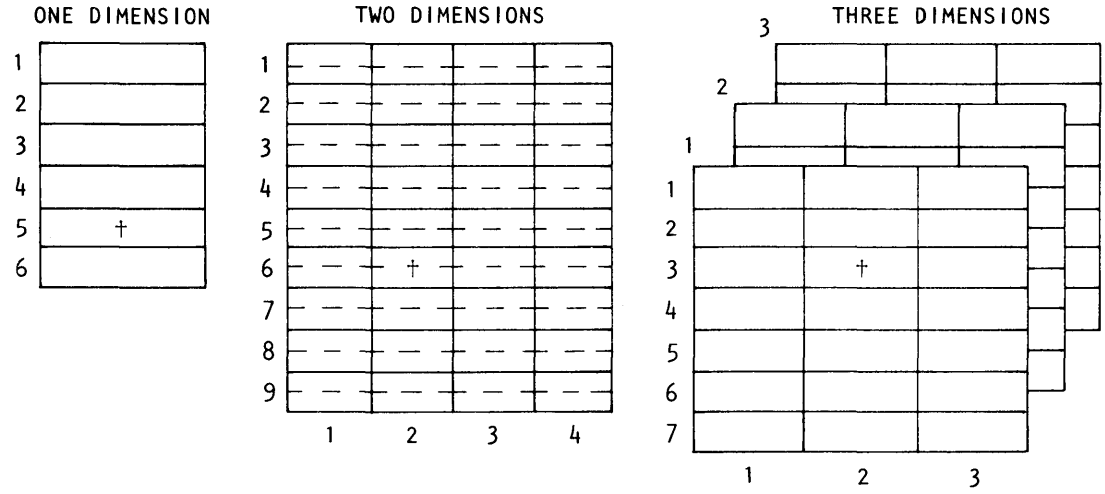
A subscript expression must yield an integer value when evaluated and can contain references to integer or Boolean constants, variables, functions, and array elements. The evaluation of the subscript expression must not alter the value of other expressions within the same statement.

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

ARRAY STORAGE SEQUENCE

An array has a storage sequence defined by the storage sequence of its elements. 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.

The number of elements in an array is the product of the number of elements in each dimension. Examples of array storage sequence appear in figure 2-1.



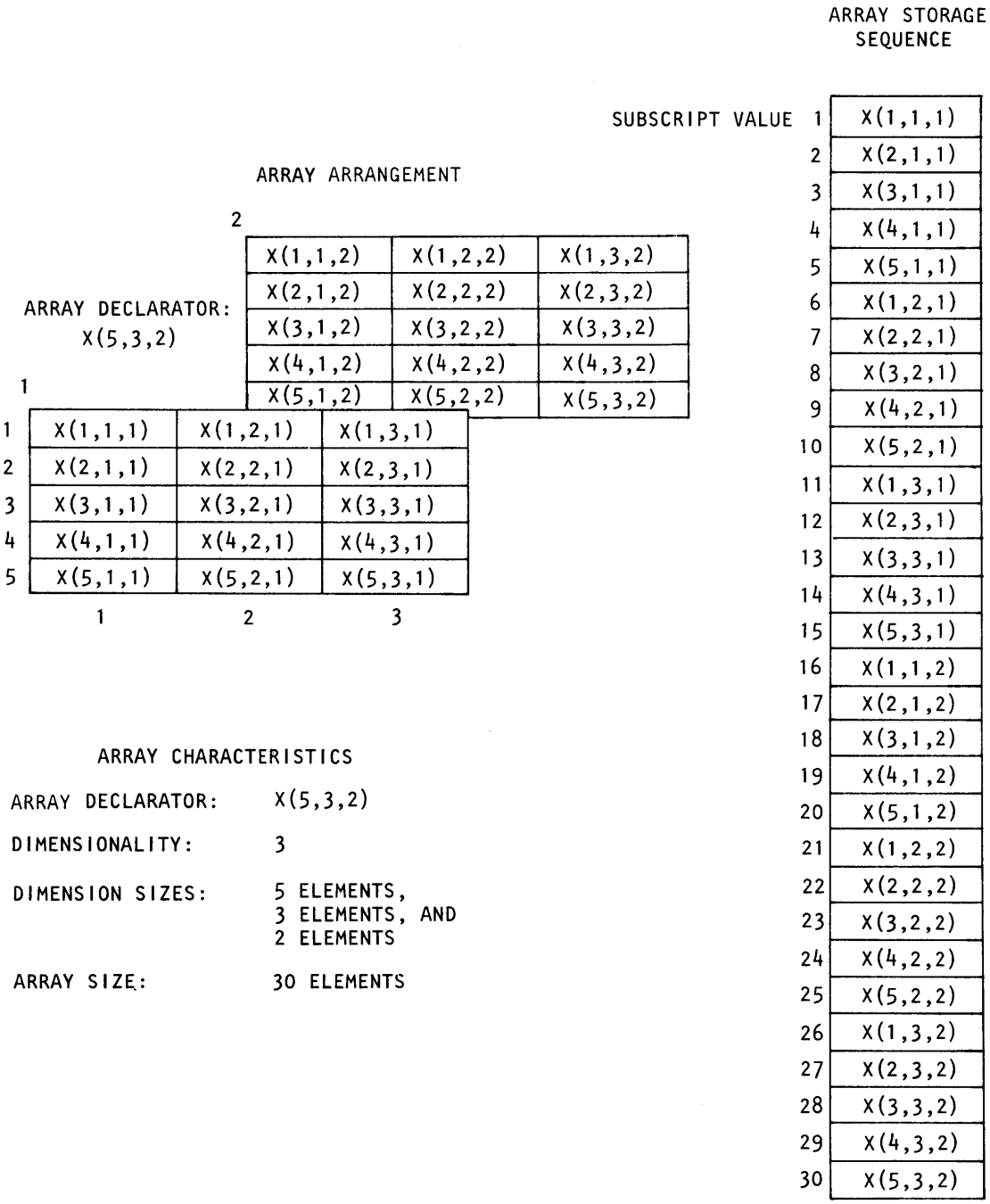
ARRAY DECLARATOR:	ARX (6)	ARY (9,4)	IARZ (7,3,3)
DATA TYPE:	REAL	DOUBLE PRECISION	INTEGER
DIMENSION SIZES:	6 ELEMENTS	9 ELEMENTS AND 4 ELEMENTS	7 ELEMENTS, 3 ELEMENTS, AND 3 ELEMENTS
TOTAL ELEMENTS:	6	36	63
† ARRAY ELEMENT REFERENCE:	ARX (5)	ARY (6,2)	IARZ (3,2,1)
NUMBER OF WORDS:	6	72	63

Figure 2-1. Array storage sequence

ARRAY ELEMENT ORDER

The subscript portion of an array element name has a value identifying its placement in that array. An array name designating an entire array implies the sequential specification of all subscripts and the processing of all elements in that order.

Array elements are arranged and referenced in terms of dimensions, but are stored in ordinal sequence in memory. (See figure 2-2).



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)

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	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(4,2,1)
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)

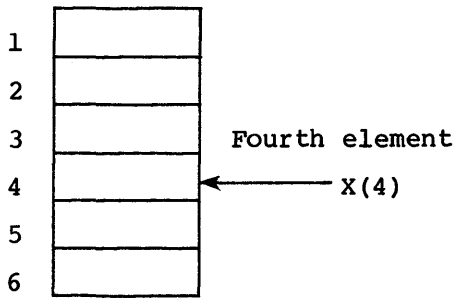
Figure 2-2. Array element arrangement and reference

SUBSCRIPT VALUES

The subscript portion of an array element name has an integer value that identifies its placement in that array. Before evaluation, a subscript expression might not be the same value as its corresponding placement in the array. Consider the following examples, each of a subscript referencing the fourth element of an array.

Example A:

X(1:6)



Example B:

Y(-2:3)

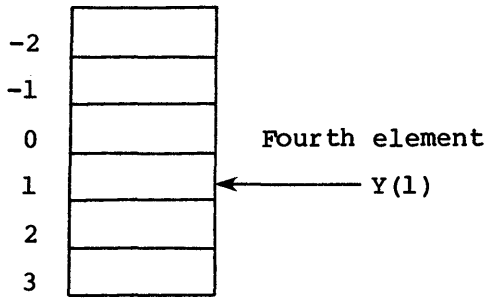


Table 2-3 illustrates the method for evaluating subscript expressions.

DUMMY AND ACTUAL ARRAYS

A *dummy array* is declared by a dummy array declarator. An *actual array* is declared by an actual array declarator. (See descriptions of dummy and actual array declarators in the subsection, Kinds of Array Declarators, earlier in this section). A dummy array is permitted only in function or subroutine subprograms. Each array in a main program is an actual array specified by a constant array declarator. Actual arrays can also be specified in function and subroutine 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. If it is an array name, the size of the dummy array must not exceed the size of the actual array. If the 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 $n-s+1$. Each dummy array must be associated through one or more levels of external procedure references with an actual array or an actual array element.

Table 2-3. Subscript evaluation

n	Dimension declarator	Subscript	Subscript value
1	$(j_1:k_1)$	(s_1)	$1+(s_1-j_1)$
2	$(j_1:k_1,j_2:k_2)$	(s_1,s_2)	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$
3	$(j_1:k_1,j_2:k_2,j_3:k_3)$	(s_1,s_2,s_3)	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$ $+(s_3-j_3)*d_2*d_1$
. . .			
n	$(j_1:k_1,\ldots j_n:k_n)$	$(s_1,\ldots s_n)$	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$ $+(s_3-j_3)*d_2*d_1$ $+\ldots$ $+(s_n-j_n)*d_{n-1}$ $*d_{n-2}*\ldots*d_1$

where:

n = Number of dimensions ($1 \leq n \leq 7$)
j = Lower bound of dimension declarator
k = Upper bound of dimension declarator
s = Subscript expression ($j_i \leq s_i \leq k_i$)
d = $(k_i-j_i)+1$

ADJUSTABLE ARRAYS AND ADJUSTABLE DIMENSIONS

An *adjustable array* is declared by an adjustable array declarator (see description of kinds of array declarators, earlier in this section) in which dimension declarators can contain variables, array elements, or functions called *adjustable dimensions*. Array elements, if specified, must not be elements of the array being declared. The name of an adjustable array must appear in a dummy argument list of a subprogram. A variable or array element that is contained in a dimension declarator must be named in the dummy argument list containing the array name or in a COMMON statement in the same subprogram.

Each actual argument corresponding to a dummy argument and each variable in common that is used in a dummy declaration must be defined with a value before being used. The values of those dummy arguments or

variables in common and any constants appearing in the dummy array declarator determine the size of the corresponding adjustable dimension for that execution of the subprogram.

The sizes of the adjustable dimensions and of any constant dimensions appearing in an adjustable array declarator determine the number and order of elements in the array. Each reference to a subprogram can define different properties (size of dimensions, number of elements, element ordering) for each adjustable array in that subprogram. These properties depend on the values of any actual arguments and variables in common when the subprogram is referenced.

Adjustable array properties of dimension number and array size do not change during subprogram execution. Variables defining an adjustable dimension can be redefined or become undefined during execution of the subprogram with no effect on these properties.

USE OF ARRAY NAMES

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.

- List of dummy arguments
- COMMON statement
- Type statement
- Array declarator
- EQUIVALENCE statement
- DATA statement
- List of actual arguments in a reference to an external procedure
- List of an input/output statement if it is not an assumed-size dummy array
- Unit identifier for an internal file in an input/output statement if it is not an assumed-size dummy array
- Format identifier in an input/output statement
- NAMELIST statement
- Pointee in a POINTER statement
- SAVE statement

CHARACTER SUBSTRINGS

A character *substring* consists of one or more contiguous characters within a character string. The substring name consists of a variable name or an array element name, followed by a *substring designator* of the format:

```
([ix]:[iy])
```

where *ix* is an integer expression that designates the first character position of the desired substring and

iy is an integer expression that designates the last character position of the desired substring.

The value of *ix* must be at least one; the value of *iy* must not be greater than the length of the string or less than *ix*. If *ix* is omitted, the substring is assumed to begin at the first position of the string. If *iy* is omitted, the substring is assumed to end at the last position. Omitting both expressions designates the entire string as a substring. Some examples of substring use follow.

Examples:

STRINGA(6:9) designates the sixth through the ninth positions of character variable STRINGA.

STRINGB(2,6)(1:3) designates the first through the third positions of array element STRINGB(2,6).

STRINGC(5,4)(:7) designates the first through the seventh positions of array element STRINGC(5,4).

STRINGD(4:) designates the fourth through the last position of character variable STRINGD.

STORAGE AND ASSOCIATION

Storage sequences describe association among variables, array elements, common blocks, and arguments.

STORAGE SEQUENCES

A *storage sequence* is a sequence of storage units. A storage unit corresponds to the Cray Computer System word of 64 bits. An integer, real, Boolean, or logical data occupies one storage unit; a double-precision or complex data occupies two storage units. A data requiring more than one storage unit in a storage sequence occupies consecutive locations in memory.

Character data are represented as 8-bit ASCII values, packed eight per word.

The ANSI FORTRAN Standard does not specify the relationship between storage units and computer words, or provide for character packing.

A storage sequence corresponds to a contiguously addressed set of memory locations.

The term storage sequence describes relationships associating variables, array elements, arrays, and common blocks.

Each array and each common block has a storage sequence. Two storage sequences are *associated* if they share at least one storage unit. The *size of a storage sequence* is the number of storage units it contains. A storage unit contains one variable or array element of type integer, real, or logical.

A double-precision or complex variable or array element has a storage sequence of two storage units. In a double-precision storage sequence, the most significant and least significant parts of data are contained in the first and second storage units, respectively.

In a complex storage sequence, the real and the imaginary parts of data are contained in the first and second storage units, respectively. The storage size for character data depends on the length specification of the data.

ASSOCIATION OF ENTITIES

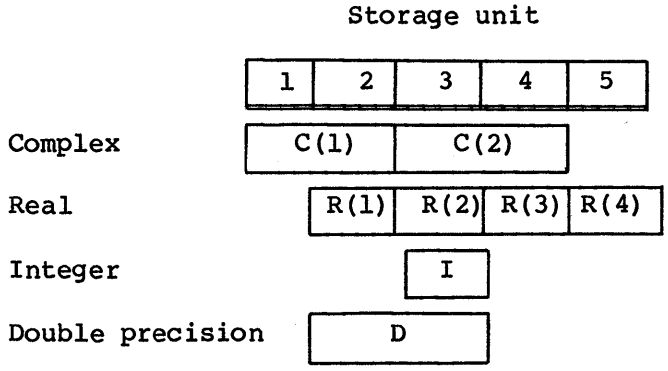
Association occurs when data can be identified by different symbolic names or from different program units. Two entities are associated if their storage sequences are associated. *Totally associated* entities share the same storage sequence. *Partially associated* entities share part but not all of a storage sequence.

Partial association can exist between a double-precision or complex entity and a second entity of type integer, real, logical, double-precision, or complex; or between two character entities. Partial association occurs only by using the COMMON, EQUIVALENCE, or 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 third storage unit of C, the second storage unit of R and the storage unit of I are specified as the same. The storage sequences can be illustrated in the following way.



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.

The definition status and value of an entity affect the definition status and value of any associated entity or entities. An EQUIVALENCE statement, a COMMON statement, or argument association in a procedure reference can cause the association of storage sequences. The association of data in two different COMMON statements is illegal.

An EQUIVALENCE statement causes association of entities within a program unit unless one of the entities is also in a common block.

Arguments and COMMON statements cause entities in two or more program units to become associated.

DEFINITION

During program execution, the content of a given variable or array element is either *defined* or *undefined*. A *defined* variable or array element contains a value. An *undefined* variable or array element does not contain a predictable value. Once defined, a variable or array element contains a specific value until it becomes undefined or is redefined with a different value.

All variables and array elements are initially undefined and remain so until action before or during program execution defines them. 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 at that point in program execution where it is required.

Defined entities

Variables and array elements become defined in the following cases.

- Execution of an arithmetic, character, or logical assignment statement causes the entity to the left of the equal sign to become defined.
- When an input statement is executed, each entity is assigned a value and thus becomes defined.
- Execution of a DO statement causes the DO variable to become defined.
- Beginning execution of actions specified by an implied-DO list in an input/output statement causes the implied-DO variable to become defined.
- A DATA statement causes entities to become initially defined when execution of a program begins.
- Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.
- When an entity of a given type becomes defined, all totally associated entities of the same type become defined. However, entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.
- A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined.

- Execution of an input/output statement that contains a status specifier causes the specified integer variable or array element to become defined.

- During the execution of an INQUIRE statement, any entity that is assigned a value becomes defined if no error condition exists.

- 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.

- When all characters of a character entity become defined, the character entity becomes defined.

Undefined entities

Variables and array elements become undefined in the following cases.

- All entities are undefined at the beginning of program execution except those entities initially defined by DATA statements.

- When an entity of a given type becomes defined, all totally associated entities of different type become undefined.

- Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Entities of type integer that are associated with the variable are also undefined as integers.

- When a noncharacter type entity becomes defined, all partially associated entities become undefined. However, when an entity of type real is partially associated with an entity of type complex, the complex entity does not become undefined when the real entity becomes defined and the real entity does not become undefined when the complex entity becomes defined. When an entity of type complex is partially associated with another entity of type complex, definition of one entity does not cause the other to become undefined.

- If the value of a function is not needed to determine the value of the expression in which the function is referenced, the function argument or the entity in common becomes undefined.

- The 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 blank common
  - Initially defined entities
  - Entities specified by SAVE statements
  - Entities in a named common block appearing in both the subprogram and another program unit that references the subprogram
- When an error condition or an end-of-file condition occurs during execution of an input statement, all items in the input list of the statement become undefined.
- Execution of a direct access input statement that specifies a record not previously written causes all input list entities to become undefined.
- Execution of an INQUIRE statement might cause entities to become undefined.
- When any character of a character entity becomes undefined, the character entity becomes undefined.
- When an entity becomes undefined as a result of conditions described in the five preceding items, all totally associated entities become undefined and all partially associated entities except those of type character become undefined.

SYMBOLIC NAMES

A symbolic name is the name of a constant, a variable, an array, a common block, a main program, a subprogram, an intrinsic function, a statement function, a block data subprogram, or a procedure. A symbolic name consists of from one to eight alphanumeric characters, the first of which must be a letter. 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.

SCOPE OF SYMBOLIC NAMES

The scope of a symbolic name is an executable program, a program unit, or a statement function 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.

Global entities

The main program, common blocks, subprograms, and external procedures are global entities of an executable program. A symbolic name that identifies a global entity 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 one of the following classes.

- Common block
- External function
- Subroutine
- Main program
- Block data subprogram

Local entities

The scope of a symbolic name of a local entity is a single program unit. A symbolic name that identifies a member in one class of entities local to a program unit must not also identify a member in another class of entities local to that same program unit. However, a symbolic name that identifies a local entity can, in a different program unit, identify an entity of any class that is either local to that program unit or is global to the executable program. A symbolic name that identifies a global entity in a program unit must not also identify a local entity in that program unit except as noted for common block and external function names in the subsection, Classes of Symbolic Names, later in this section.

A local entity is identified by a symbolic name appearing in one of the following classes.

- Array
- Variable
- Constant
- Statement function
- Intrinsic function

A symbolic name used as a dummy argument in a procedure identifies a variable, an array, or another procedure. This specification and usage must not violate the respective class rules.

CLASSES OF SYMBOLIC NAMES

In a program unit, a symbolic name must not correspond to more than one class except as noted in the following paragraphs. All restrictions on the appearances of the same symbolic name in different program units of an executable program are also noted here.

Common blocks

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 can also be the name of a local entity other than a constant, intrinsic function, 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 only identifies the local entity.

External functions

A symbolic name is the name of an external function if one of the following conditions exists.

- The name appears immediately following the keyword FUNCTION or ENTRY in a FUNCTION or ENTRY statement.

- The name is not an array name, statement function name, intrinsic function name, subroutine name, or dummy argument and every appearance is followed by a left parenthesis except in a type statement, in an EXTERNAL statement, or as an actual argument.

The name of a function subprogram that appears immediately after the keyword FUNCTION or ENTRY in a FUNCTION or ENTRY statement must be the name of a variable in that subprogram. An external function name is global to the executable program.

Subroutines

A symbolic name is the name of a subroutine if one of the following conditions exists.

- The name appears immediately following the keyword SUBROUTINE or ENTRY in a SUBROUTINE or ENTRY statement.

- The name appears immediately following the keyword CALL in a CALL statement and is not a dummy argument.

A subroutine name is global to the executable program.

The main program

A symbolic name is the main program name if it appears in a PROGRAM statement in the main program. A main program name is global to the executable program.

Block data subprograms

A symbolic name is the name of a block data subprogram if it appears in a BLOCK DATA statement. A block data subprogram name is global to the executable program.

Arrays

A symbolic name is the name of an array if it appears as the array name in an array declarator in a DIMENSION, COMMON, or type statement. An array name is local to a program unit and can be the same as a common block name.

Variables

A symbolic name is the name of a variable if it meets all of the following conditions.

- The name does not appear in a PARAMETER, INTRINSIC, or EXTERNAL statement.
- The name is not the name of one of the following.
  - An array
  - A subroutine
  - A main program
  - A block data subprogram
- The name appears as a name other than one of the following.
  - An external function in a FUNCTION statement
  - A common block
  - An entry name in an ENTRY statement in an external function
- The name is not immediately followed by a left parenthesis unless one of the following conditions exists.
  - It is immediately preceded by the word FUNCTION in a FUNCTION statement.
  - It is immediately preceded by the word ENTRY in an ENTRY statement.
  - It is at the beginning of a character substring name.

A variable name can be a parameter enclosed in parentheses in a FUNCTION statement. (See the FUNCTION statement in part 2, section 7.)

A variable name is local to a program unit. A variable name in the dummy argument list of a statement function statement is local to the statement function statement where it occurs.

A statement function dummy argument name can also be the name of a variable or common block in the same program unit. The appearance of the name in any context other than as a dummy argument of the statement function identifies a local variable or common block. The statement function dummy argument name and local variable name have the same type. If the type is character, they also have the same length. A variable can have the same name as a common block.

Constants

A symbolic name is the name of a constant if it appears as a symbolic name in a PARAMETER statement. A constant name is local to a program unit. A constant can have the same name as a common block.

Statement functions

A symbolic name is the name of a statement function if it is not an array name and if a statement function statement specifies that symbolic name. A statement function name is local to a program unit. A statement function name can be the same as a common block name.

Intrinsic functions

A symbolic name is the name of an intrinsic function if the following conditions exist.

- The name is not an array name, statement function name, subroutine name, or dummy argument name.
- Every appearance of the symbolic name, except in a type statement, an INTRINSIC statement, or as an actual argument, is immediately followed by an actual argument list enclosed in parentheses.

An intrinsic function name is local to a program unit. (See Appendix B for intrinsic function list.)

Dummy procedures

A symbolic name is the name of a dummy procedure if the name appears in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement and meets one or more of the following conditions.

- It appears in an EXTERNAL statement.
- It appears as the name of the called subroutine in a CALL statement.
- It is not an array name or character variable name and it is immediately followed by a left parenthesis, except in the following cases.
  - In a type statement
  - In an EXTERNAL statement
  - In a CALL statement
  - As a dummy argument
  - As an actual argument
  - As a common block name in a COMMON or SAVE statement

A dummy procedure name is local to a program unit.

NAMELIST group name

A NAMELIST group name names the list that follows the group name. The group name must be unique 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
```

EXPRESSIONS 3

An expression calls for the evaluation of one or more operands. Expressions can include operators and parentheses to specify the manner and order of their evaluation in yielding a single value. Operands can be constants, symbolic names of constants, variables, array elements, substrings, and function references. Operators specify the arithmetic, character, relational, or logical operations to be performed on these operands.

Expressions are one of the following types.

- Arithmetic
- Character
- Relational
- Logical
- Boolean

The ANSI FORTRAN Standard does not provide for Boolean expressions.

ARITHMETIC EXPRESSIONS

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. Only exponents of type integer are permitted. References to variables, array elements, or functions are not permitted.

An *arithmetic expression* specifies a numeric computation. Its evaluation produces a single numeric 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. Arithmetic operands are of type integer, real, double precision, or complex.

ARITHMETIC OPERATORS

The arithmetic operators are given in table 3-1.

Table 3-1. Arithmetic operators

Operator	Operation
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or negation
+	Addition or identity

Each arithmetic operator operates on a pair of operands and appears between them. In addition, either of the operators + and - can operate on a single operand when it precedes that operand.

Interpretation of arithmetic operators in expressions

The interpretation of expressions formed with each arithmetic operator is shown in table 3-2. (X and Y are operands.)

Table 3-2. Interpretation of operators 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)

The interpretation of a division operation might depend on the data types of the operands as described in part 1, section 2.

Precedence of arithmetic operators

Quantities enclosed in parentheses are evaluated first. If parentheses are within parentheses, the innermost quantity is evaluated first. Then the operations are evaluated according to the precedence shown in table 3-3.

Table 3-3. Precedence of arithmetic operators

Operator	Precedence
**	First
* and /	Second
+ and -	Third

For example, in the expression

```
-A**2
```

the exponentiation operator (**) has precedence over the negation operator (-). Therefore, the operands of the exponentiation operator are combined and then used as the operand of the negation operator. Thus, the interpretation of the above expression is the same as the mathematical interpretation of the expression

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

When an expression involves two or more operations on the same precedence level, their position within the expression determines the order of their evaluation.

ARITHMETIC OPERANDS

Arithmetic operands are:

- Primaries,
- Factors,

- Terms, and
- Arithmetic expressions.

The following subsections describe the forms of combining operands and operators in arithmetic expressions.

Primaries

Primaries are:

- Unsigned arithmetic constants,
- Symbolic names of arithmetic constants,
- Variable references,
- Array element references,
- Function references, and
- 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

Factors

The forms of a *factor* follow.

- Primary
- Primary ** factor

Thus, a factor is a sequence of one or more primaries with its elements separated by the exponentiation operator. The second form indicates that in 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
```

has the same interpretation as the factor

```
2**(3**2).
```

Terms

The forms of a *term* follow.

- Factor
- Term / factor
- Term * factor

Thus a term is a factor or a sequence of factors with its elements separated by a multiplication or a division operator. The last two forms indicate that the factors are combined from left to right in interpreting a term containing two or more multiplication or division operators.

Arithmetic expressions

The forms of an *arithmetic expression* follow.

- Term
- + term
- - term
- Arithmetic expression + term
- Arithmetic expression - term

Thus, an arithmetic expression is a term or a sequence of terms with its elements separated by an addition (+) or a subtraction (-) operator. The first term in an arithmetic expression can be preceded by an identity (+) or negation (-) operator. The last two forms imply that terms are combined from left to right in interpreting an arithmetic expression containing two or more addition or subtraction operators.

These formation rules do not permit expressions containing two consecutive arithmetic operators such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

DATA TYPE OF ARITHMETIC EXPRESSIONS

The form of a constant determines its data type. The data type of a named constant, variable, array element, or function reference is determined by its name. The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands.

Integer expressions, *real expressions*, *double-precision expressions*, *complex expressions*, and *Boolean expressions* are arithmetic expressions that have values of type integer, real, double precision, complex, and Boolean, respectively.

When a + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand.

The data types of arithmetic expressions are given in table 3-4. In this table, each letter designates the type of operand or result as integer (I), real (R), double precision (D), complex (C), logical (L), or Boolean (B).

To use the table, locate the types of the first and second operands in the first and second columns, respectively. The third column contains the type of the expression formed when these operands are processed by an arithmetic operator or when the expression contains only one operand. In an expression, the type of the operand with parentheses indicates the type of the result. For example, in the expression

```
D(1,2,3) * I
```

the integer variable I with the double-precision array element D(1,2,3) yields a result of type double-precision. As shown in table 3-4, if two operands are of different types, the one differing in type from the result type is first converted to the type of the result, then the operation is performed.

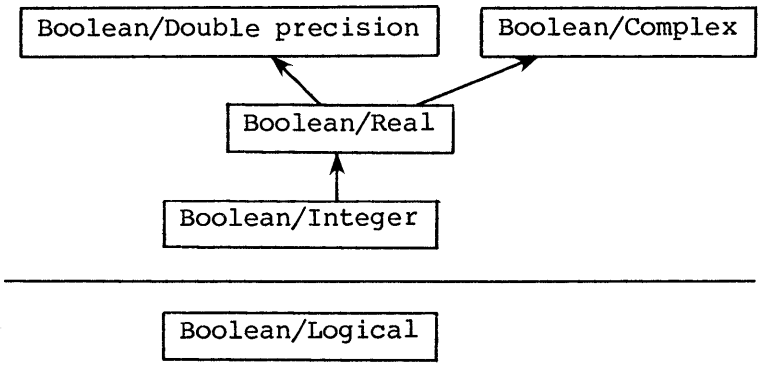
Table 3-4. Arithmetic operand, expression, and result typing relationships

x @ y→z	x @ y→z	x @ y→z	x @ y→z	x @ y→z	x @ y→z
I I I	R (I) R	D (I) D	C (I) C	L I †	B I I
(I) R R	R R R	D (R) D	C (R) C	L R †	B R R
(I) D D	(R) D D	D D D	C (D) C	L D †	(B) D D
(I) C C	(R) C C	(D) C C	C C C	L C †	(B) C C
I L †	R L †	D L †	C L †	L L L*	B L L*
I B I	R B R	D (B) D	C (B) C	L B L*	B B B**

Legend:

x,y	Arithmetic operands
@	Arithmetic operator
x @ y	Arithmetic expression or single arithmetic operand
z	Arithmetic result
( )	Conversion required before computation
†	Prohibited
*	@ must be a logical operator (for example, .AND.)
**	Arithmetic is done as if the operands were integer

Conversion is always upward; that is, type hierarchy determines which operand is converted. The following example shows type hierarchy.



In the preceding example, type Boolean appears on all levels and needs no conversion. The exception is at the highest level, Boolean must be converted because it must be extended for double precision, and the imaginary portion must be established for type complex.

The lowest level is not upward convertible because arithmetic operations with logical operands are prohibited. All other levels can be converted to any higher level.

Table 3-5 shows the conversion that takes place when the equal sign is processed in assignment statements.

Table 3-5. Type conversion in assignment statements

x \ y	I	R	D	C	L	B	CH
I	n/c	(R)	(D)	(C)	*	n/c	*
R	(I)	n/c	(D)	(C)	*	n/c	*
D	(I)	(R)	n/c	(R)†	*	(B)	*
C	(I)	(R)	(R)†	n/c	*	(B)	*
L	*	*	*	*	n/c	n/c	*
CH	*	*	*	*	*	*	n/c

Legend:

x=y	Assignment statement
( )	Conversion required before assignment to result
*	Prohibited
n/c	No conversion necessary

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. In particular, the type and interpretation of such an expression are independent of the type of any other operand of any larger expression where it appears.

Integer quotients

An *integer quotient* is the integer portion of a mathematical quotient having an integer divisor and dividend. For example, the expression -5/2 yields an integer quotient of -2.

† When C=D, D is converted to REAL and assigned to the real part of C; the imaginary part of C=0.0. When D=C, the real part of C is converted to double precision; the imaginary part of C is not used in the conversion.

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. Type conversion is based on the following two operations.

(a) Integer-to-real conversion creates a real value from an integer value. The maximum 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 maximum absolute value of the number being converted must be less than 2**46. The fractional part is truncated. No warning is issued if the value exceeds the range.

Type integer - Type integer conversion to:

- Type real occurs as described in (a) above.
- Type double-precision occurs as described in (a) above and with zeros established as the extended portion of the value.
- Type complex occurs as described in (a) above. The result becomes the real portion of the complex value and zero is established as the imaginary portion.

Type real - Type real conversion to:

- Type integer occurs as described in (b) above.
- Type double-precision is accomplished by extending the precision of the real value through the addition of zeros as the least significant portion.
- Type complex is accomplished by establishing the real value as the real portion of the complex value and by establishing zero in the imaginary portion.

Type double-precision - Type double-precision conversion to:

- Type integer occurs by connecting the most significant portion by (b) above.
- Type real is accomplished by establishing the most significant portion as the real value. No rounding occurs.
- Type complex is accomplished by establishing the most significant portion as the real portion of the complex value, and establishing zero as the imaginary portion of the complex value. No rounding occurs.

The ANSI FORTRAN Standard does not provide for double-precision to complex conversion.

Type complex - Type complex conversion to:

- Type integer is accomplished by converting the real portion of the complex value as described for the real value in (b) above.
- Type real is accomplished by establishing the real portion of the complex value as the real value.
- Type double-precision is accomplished by extending the precision of the real portion of the complex value as for real.

Type Boolean - Type Boolean conversion to:

- Types integer, real and logical is accomplished with no change in its bit pattern.
- Type double-precision extends the precision by establishing zero as the least significant portion.
- Type complex uses zeros for the imaginary portion and establishes the real portion as the bit pattern of the Boolean value.

EVALUATION OF ARITHMETIC EXPRESSIONS

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, because of finite approximation to real numbers and round-off errors, mathematically equivalent arithmetic expressions can produce results that differ computationally.

The difference between the values of the expressions 5/2 and 5./2. is mathematical and is not a computational difference. The difference between 5./10. and 5.*.1 is a computational difference.

In addition to parentheses required for the desired interpretation, other parentheses can be included to control the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

```
A+(B-C)
```

the term (B-C) is evaluated and then added to A. Including parentheses could change the computational value.

For example, the two expressions

```
A*I/J
```

```
A*(I/J)
```

might have different computational values if I and J are integer factors.

CHARACTER EXPRESSIONS

A *character primary* is a character constant, a symbolic name of a character constant, or a variable, an array element, a substring, or a function reference. A *character expression* is a sequence of one or more character primaries. If more than one character primary comprises the expression, the primaries are concatenated by the character operator, the double slash.

CHARACTER EXPRESSION EVALUATION

The result of character expression evaluation is always of type character. Primaries are combined from left to right.

Example:

The sequence

```
CHARACTER*3 VAR1,VAR2
VAR1='CRA'
VAR2='Y-1'
      .
      .
      .
PRINT *,VAR1//VAR2
```

produces the printed result

```
CRAY-1
```

When used in an arithmetic or noncharacter relational expression, a character constant of length less than or equal to 8 is considered to be type Boolean. A length greater than 8 is illegal in an arithmetic or noncharacter relational expression.

RELATIONAL EXPRESSIONS

A relational expression compares the values of two arithmetic or character expressions, producing a type logical value of true or false.

Two relational expressions are relationally equivalent if their logical values are equal for all possible values of their primaries.

Relational expressions can appear within logical expressions.

Relational operators are shown in table 3-6.

Table 3-6. Relational operators

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.

ARITHMETIC RELATIONAL EXPRESSIONS

The form of an *arithmetic relational expression* is

e_1 *relop* e_2

where e_1 and e_2 are each integer, real, double precision, Boolean, or complex expressions and

relop is a relational operator.

A complex expression is permitted only when the relational operator is .EQ. or .NE.

An arithmetic relational expression is interpreted as logical value true if the values of the expressions satisfy the relation specified by the operator; false if they do not.

If the two arithmetic expressions are of different types (see table 3-4), the types of the operands are converted as if the expression were

$((e_1)-(e_2))$ *relop* 0.

Examples:

```
INDEX .EQ. ENDVALU

J(1,6,6)*COS(ALPHA/10.) .GT. Z

A .LE. B

3.1415927 .LT. (22./7.)

CMPLXM .NE. COMPLXN
```

CHARACTER RELATIONAL EXPRESSIONS

The form of a *character relational expression* is

e_1 *relop* e_2

where e_1 and e_2 are character expressions and

relop is a relational operator. (See table 3-6 for a list of relational operators.)

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 the one considered to be of less 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.

LOGICAL EXPRESSIONS

A logical expression expresses a logical computation. Evaluation of a logical expression produces a result of type logical with a value of either true or false.

Two logical expressions are logically equivalent if their values are equal for all possible values of their primaries.

LOGICAL OPERATORS

Table 3-7 presents the *logical operators* and their order of precedence. The logical operators .XOR., .X., and .NEQV. perform the same logical operation.

Table 3-7. Logical operators

Operator	Operation	Precedence
.NOT. or .N.	Logical negation	First
.AND. or .A.	Logical conjunction	Second
.OR. or .O.	Logical inclusive disjunction	Third
.XOR. or .X. or .NEQV.	Logical exclusive disjunction or logical non-equivalence	Fourth
.EQV.	Logical equivalence	

FORM AND INTERPRETATION OF LOGICAL EXPRESSIONS

For logical expressions that contain two or more logical operators, the precedence determines the order in which they are to be combined (unless changed by the use of parentheses). For example, in the expression

```
A .OR. B .AND. C
```

the .AND. operator has higher precedence than the .OR. operator. Therefore, the interpretation is the same as the interpretation of

A .OR. (B .AND. C).

The ANSI FORTRAN Standard does not provide for the .XOR. operator or for .N., .A., .O., or .X. as abbreviations.

Logical operands include the following.

- Logical primaries
- Logical factors
- Logical terms
- Logical disjuncts
- Logical expressions

The following paragraphs describe the forms of combining operands and operators in logical expressions.

Logical primaries include 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

The following is the form of a *logical factor*.

- [.NOT.] logical primary

The following is the form of a *logical term*.

- [logical term .AND.] logical factor

Thus, 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 following is the form of a *logical disjunct*.

- [logical disjunct .OR.] logical term

A logical disjunct is, therefore, 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 following forms are *logical expressions*.

- [logical expression .XOR.] logical disjunct
- [logical expression .EQV.] logical disjunct
- [logical expression .NEQV.] logical disjunct

A logical expression is, therefore, 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.

These forms allow the logical operator .NOT. to immediately follow any other logical operator. For example, the logical term

```
LOGICALX .AND. .NOT. LOGICALY
```

is permitted.

VALUES OF LOGICAL FACTORS, TERMS, DISJUNCTS, AND EXPRESSIONS

The value of a logical factor involving .NOT. or .N. is shown below.

X	.NOT. X
true	false
false	true

Logical values involving .AND., .OR., .XOR., .NEQV., and .EQV. are shown below.

X_1	X_2	X_1.AND.X_2	X_1.OR.X_2	X_1.XOR.X_2 X_1.NEQV.X_2	X_1.EQV.X_2
true	true	true	true	false	true
true	false	false	true	true	false
false	true	false	true	true	false
false	false	false	false	false	true

BOOLEAN (MASKING) EXPRESSIONS

A Boolean expression is evaluated to yield a string of 64 binary digits representing bit positions in a storage unit (64-bit Cray computer word). The forms of Boolean expressions appear below.

- Boolean constant
- Name of a function providing a Boolean data when referenced
- Boolean expression of the form B_1 *lop* B_2 where *lop* is one of the logical operators in table 3-7 and B_1 and B_2 are not complex, double-precision, or logical expressions
- Boolean expression enclosed in parentheses

The ANSI FORTRAN Standard does not provide for Boolean expressions.

Boolean expressions can be combined with expressions of Boolean or other types by using arithmetic, relational, and logical (masking) operators. Boolean data is never converted to a different type. Evaluation of an arithmetic or relational operator processes a Boolean expression with no type conversion. If both operands of an arithmetic or relational operator are Boolean, they are processed as if they were integer and the result is of type Boolean or logical, respectively. Two Boolean expressions are equivalent if their values are equal for all possible values of their primaries.

A logical (masking) operator processing a Boolean expression performs a bit-by-bit logical (masking) operation. The result of the operation is of type Boolean. The interpretation of Boolean factors, terms, and expressions is the same as described in the subsection Logical Expressions, earlier in this section. The results of binary one and zero correspond to the logical results of true and false, respectively, in each of 64 bit positions. These values are summarized in the following chart.

X_1	X_2	.NOT.X_1	X_1.AND.X_2	X_1.OR.X_2	X_1.XOR.X_2 X_1.NEQV.X_2	X_1.EQV.X_2
1100	1010	0011	1000	1110	0110	1001

PRECEDENCE OF ALL OPERATORS

Precedence among all types of operators is presented in table 3-8.

Table 3-8. Precedence among all operators

Operator	Precedence
Arithmetic	First
Character	Second
Relational	Third
Logical	Fourth

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

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

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

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

EVALUATION OF EXPRESSIONS

A variable, array element, or function referenced as an operand in an expression must be defined at the time the reference is executed. Names of constants must be established in a PARAMETER statement preceding the statement of first reference.

An arithmetic operation with a result that cannot be mathematically defined produces unpredictable results in an executable program. Each term of an expression is evaluated even if some terms are not needed to determine the result. For example, the logical expression

```
(A.EQ.0).OR.(B/A.GT.10)
```

causes a divide fault if A=0. Expressions that raise 0 to a 0 or negative power or that raise a negative value to a non-integer power also cause run-time faults.

The execution of a function reference in a statement must not alter the value of any other entity within the same statement. Nor may it alter the value of any entity in common that affects the value of any other function reference in that same statement. If a function reference in a statement causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A(I)=F(I) and

Y=G(X)+X
```

where F and G are functions, produce unpredictable results when the reference to F defines I or the reference to G defines X.

The data type of an expression in which a function reference appears neither affects nor is affected by the evaluation of the actual arguments of the function.

The data type of an expression in which an array element is referenced neither affects nor is affected by the evaluation of the subscript.

ORDER OF EVALUATION OF FUNCTIONS

The order of evaluation of multiple function references within a single statement is fixed only within a direct logical IF statement and within nested function references.

Examples:

- In the statement IF(F(Y))A=F(Y) where F is a function name, the function reference in the conditional statement A=F(Y) is evaluated last.

- In the statement A=F(G(X)) where F and G are functions, G is evaluated first.

In other statements that contain more than one function reference, the value provided by each function reference must not be affected by the order in which the other function references are evaluated.

PARENTHESES AND EXPRESSIONS

A parenthesized expression is treated as an entity. For example, in evaluating the expression A*(B*C), the product of B and C is evaluated and then multiplied by A. Parenthesized expressions can contain one or more parenthesized expressions, each of which can contain one or more parenthesized expressions, etc. This nesting of parenthesized expressions can be specified to 63 levels.

The ANSI FORTRAN Standard does not limit the number of levels of nested parentheses.

SUMMARY OF RULES OF INTERPRETATION

The order in which primaries are combined using operators is determined by the following conditions.

- 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 Boolean term

- Left-to-right interpretation of inclusive disjunctions in a logical or Boolean disjunct

- Left-to-right interpretation of exclusive disjunctions, equivalences and non-equivalences in a logical or Boolean expression

SUBROUTINE, FUNCTION, AND SPECIFICATION SUBPROGRAMS

4

Subprograms are procedures, either predefined or supplied with the main program. Usually a subprogram contains a sequence of steps needed more than once in a main program or needed by many programs. Subprograms can also provide a means of modularizing a program; that is, a program can consist of a series of subprograms.

The two types of subprograms are specification and procedure. Specification subprograms are non-executable; with the exception of statement functions, procedure subprograms are executable.

SPECIFICATION SUBPROGRAMS

The only form of a *specification subprogram* is the block data subprogram. *Block data subprograms* provide initial values for variables and array elements in named common blocks. The block data subprogram must begin with a BLOCK DATA statement and end with an END statement. The only other statements that can appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, SAVE, DATA, and type statements. (See part 2, section 2 for format of the above mentioned statements.)

NAMED COMMON BLOCKS

A named common block can be specified in more than one block data subprogram in an executable program.

If a named common block initializes an entity, all entities having storage units in the common block storage sequence must be specified even if they are not all initialized. More than one named common block can have entities initialized in a single block data subprogram. Entities not in a named common block must neither be initialized nor appear in a DIMENSION, EQUIVALENCE, or type statement in a block data subprogram.

PROCEDURE SUBPROGRAMS

Procedure subprograms are of two types: subroutine subprograms and function subprograms. Both types are executable, but differ in the manner in which they are defined and referenced.

SUBROUTINE SUBPROGRAMS

A *subroutine subprogram* is a sequence of executable code referenced from a main program or a procedure subprogram. A subroutine must not reference itself, directly or indirectly. The first statement must be a SUBROUTINE statement; the last line must contain an END statement. A subroutine subprogram can contain one or more ENTRY statements.

A subroutine subprogram must be referenced with a CALL statement in the referencing program unit. When the CALL statement is executed, the referenced subroutine must be one of the subroutines in the executable program.

One or more dummy arguments of a subroutine subprogram can become defined or redefined to return results. Entities specified in a COMMON statement in the subroutine can also be defined for this purpose.

Actual arguments

The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is permitted. This use is an exception to the rule requiring agreement of type since there is no type associated with either a subroutine name or an alternate return specifier.

An actual argument in a subroutine reference must be one of the following.

- An expression, except a character expression involving concatenation of an operand with a length specification of (*), unless the operand is the symbolic name of a constant
- An array name
- An array element name
- A character substring name
- An intrinsic function name

- An external procedure name
- An alternate return specifier

An actual argument in a subroutine reference can be a dummy argument appearing in a dummy argument list within the subprogram containing the reference.

Subroutine subprogram restrictions

A subroutine subprogram can contain any statement except a BLOCK DATA, FUNCTION, PROGRAM, or a second SUBROUTINE statement.

The symbolic name of a subroutine or a subroutine entry is a global name and must not be the same as another global name or local name in the referencing program unit. The referencing program unit cannot use a subroutine or subroutine entry name as an external function or external function entry name.

In a subroutine subprogram, the symbolic name of a dummy argument is local and cannot appear in an EQUIVALENCE, PARAMETER, DATA, or COMMON statement. The symbolic name of a dummy argument can be the same as a common block name. A character dummy argument with a length specification of (*) must not appear as an operand for concatenation, except in a character assignment statement.

FUNCTION SUBPROGRAMS

A *function subprogram* is a sequence of executable code which can be referenced from a main program or a procedure subprogram. A function subprogram, unlike a subroutine subprogram, is referenced by the appearance of its identifier in certain types of statements. Function subprograms can be statement functions, external functions, or intrinsic functions.

Statement functions

A *statement function* is specified by a single statement similar in form to an arithmetic, logical, or character assignment statement. This statement function definition statement can only appear after the specification statements and before the first executable statement of the program unit in which it is referenced. Since it is not a part of the normal execution sequence, a statement function definition statement is classified as a nonexecutable statement.

Statement functions can be specified within a main program, a function subprogram, or a subroutine subprogram. A statement function can only be referenced from a statement within the same program unit containing its specification. (See part 2, section 7 for the format of a statement function definition statement.)

Referencing statement functions - A statement function is referenced by using its function reference as a primary in an expression.

Execution of a statement function reference results in the following actions.

- Evaluation of actual arguments that are expressions
- Association of actual arguments with corresponding dummy arguments
- Evaluation of the statement function expression

The resulting value is used in the expression containing the function reference.

The actual arguments must agree in order, number, and type with the corresponding dummy arguments. An actual argument can be any expression except a character expression involving concatenation of an operand with a length specification of (*), unless the operand is the symbolic name of a constant.

Statement function restrictions - A statement function can be referenced only in the program unit with the statement function definition statement.

A statement function definition statement can reference another statement function preceding the reference. The symbolic name identifying a statement function cannot appear as a symbolic name in any specification statement except a type statement (to specify the type of the function) or as a common block name in the same program unit.

An external function reference (see next subsection, External functions) in the expression of a statement function definition statement must not cause a dummy argument of the statement function to become undefined or redefined.

The symbolic name of a statement function is a local name and cannot be the same as another entity name in the program unit except a common block name. The symbolic name of a statement function cannot be an actual argument and cannot appear in an EXTERNAL statement.

A statement function definition statement in a function subprogram cannot reference that function subprogram.

External functions

An *external function* is a procedure specified by a function subprogram or some other means. An external function is specified external to the program unit that references it. An external function can be specified by an EXTERNAL statement or can be implied by its usage. It can also contain one or more ENTRY statements.

Referencing external functions - An external function is referenced by using its name as a primary in an expression. A reference to an entry in a function can be similarly used.

Execution of external function references - Execution of an external function reference or a reference to an external function entry results in the following actions.

- Evaluation of actual arguments that are expressions
- Association of actual arguments with the corresponding dummy arguments
- Actions specified by the referenced function

The type of the function or function entry name in the reference must be the same as the type of the function or entry name in the referenced function. The length of the character function in a character function reference must be the same as the length of the character function in the referenced function.

Actual arguments for external functions - The actual arguments in an external function reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function or function entry. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type because subroutine names do not have a type. The subroutine name must be declared external.

An actual argument in an external function reference must be one of the following.

- An expression, except a character expression involving concatenation of an operand with a length specification of (*), unless the operand is the symbolic name of a constant
- An array name
- An array element name
- An intrinsic function name
- An external procedure name

Intrinsic functions

Intrinsic functions are commonly-used operations having prespecified identities and functions. An intrinsic function can be referenced by a main program or a procedure subprogram. The entire set of operations specified in the ANSI FORTRAN Standard is included, as well as a set of Cray FORTRAN extensions. Their specific names, generic names, function definitions, and types of arguments and results appear in Appendix B. Cray FORTRAN also provides for a set of utility procedures. These procedures, which are referenced like intrinsic functions, are described in the following subsection and in Appendix C. They perform utility operations not specified in the ANSI Standard.

Referencing intrinsic functions - An intrinsic function is referenced by using its name as a primary in an expression. The resulting value is available to the expression containing the function reference.

Many intrinsic functions accept arguments of more than one type and return a result type depending on the argument type. Generic names have been assigned to families of intrinsic functions performing similar operations but requiring different types of arguments and results. Generic names simplify the referencing of intrinsic functions because the same function name can be used with more than one type of argument. Generic names, however, cannot be used when an intrinsic function is an actual argument.

The actual arguments constituting the argument list must agree in type, number, and order with those described in Appendix B and can be any expression of the specified type. An actual argument cannot be a character expression involving concatenation of an operand with a length specified as (*) unless the operand is the symbolic name of a constant.

Intrinsic function restrictions - The result of the function becomes undefined with arguments that cause undefinable results or when the result exceeds the maximum numeric representation permitted.

Examples:

```
(If A>2**8191 and B<-2**-8191):
AMEDIAN=(AMIN1(A,B,C,D)+AMAX1(A,B,C,D))/2.0

T = TAN(THETA)
```

Utility procedures - The Cray FORTRAN programmer can reference a number of predefined functions, subroutines, and other procedures that are described in Appendix C. These *utility procedures* extend program control capabilities in the following areas.

- Cray Operating System (COS) features
- Input/output operations

Function subprogram restrictions

A function subprogram can contain any statement except a BLOCK DATA, SUBROUTINE, PROGRAM, or a second FUNCTION statement.

The symbolic name of an external function or external function entry is a global name and cannot be the same as another global name. In a referencing program unit, an external function or external function entry name cannot be used as the subroutine name in a CALL statement.

The symbolic name of a function specified by a FUNCTION statement cannot appear in another nonexecutable statement except for a type statement and must only appear as a variable in executable statements.

If the type of a function is specified in a FUNCTION statement, the function name cannot appear in a type statement. (Redundant type specifications are not allowed.)

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

A function specified by a subprogram can be referenced within another procedure subprogram or the main program of the executable program. A function subprogram cannot directly or indirectly reference itself.

The symbolic name of a function subprogram must appear as a variable name in the function subprogram. During every execution of the subprogram, this variable must become defined and, once defined, can be referenced or become redefined. The value of the function is the value of this variable when a RETURN or END statement is executed in the subprogram. The type of this value is implicit to the function name unless INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, or LOGICAL is specified to cause it to be overridden.

A function subprogram can define one or more of its dummy arguments to return values in addition to the value of the function. However, this redefinition must not affect any entities referenced on the line referencing the function.

An actual argument in a function reference can be a dummy argument appearing in a dummy argument list within the subprogram containing the reference.

The result type of a statement function or an external function reference is the same as the function name type and is specified the same as variables and arrays. The result type of each intrinsic function is specified in Appendix B. Each argument type and the number of actual arguments specified in a function reference must agree with the (dummy) arguments defined in the specification of the referenced function.

If a function subprogram name is of type character, each entry name in the function subprogram must be of type character. If the function subprogram name or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

A character dummy argument with a length specification of (*) cannot appear as an operand for concatenation, except in a character assignment statement.

Execution of function references

A function reference appears only as a primary in an arithmetic, character, or logical expression. Execution of a function reference in an expression causes the evaluation of the function identified by the symbolic name of the function subprogram.

Return of control from a referenced function completes execution of the function reference. The value of the function is then available to the expression containing the reference and being evaluated.

Referencing functions

A function is referenced in an expression and supplies a value to the expression. This value is the value of the function at the time the expression containing its reference is evaluated.

An intrinsic function can be referenced in the main program or in any procedure subprogram of an executable program.

A statement function can be referenced only in the program unit in which the statement function statement appears.

An external function can be referenced by function or entry name within another procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

Using the ENTRY statement, a procedure subprogram can be entered 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. (See part 2, section 7 for the format of the ENTRY statement.)

If a character function is referenced in a program unit, the function length specified in the program unit must be an integer constant expression.

Non-FORTRAN subprograms

A *non-FORTRAN subprogram* is a set of executable code that functions the same as a subroutine or a function subprogram. It is prepared by some means other than FORTRAN. Typically, the non-FORTRAN subprogram is written in Cray assembly language (CAL), in a high-level language other than FORTRAN, or in a version of FORTRAN not compatible with the one in use. They are separately compiled or assembled and are available in binary form upon reference during program execution.

The Macros and Opdefs Reference Manual, CRI publication SR-0012 describes the creation of non-FORTRAN subroutine subprograms using CAL and the method for programming non-FORTRAN function and subroutine subprograms using CAL.

ARGUMENTS

Arguments provide a means of communication between a referencing program unit and a referenced procedure.

Data can be communicated to a statement or intrinsic function by an argument list. Data can be communicated to and from an external procedure by an argument list or by common blocks. (See subsection, COMMON BLOCKS, later in this section.) Procedure names can be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference.

The number, type, and order of actual arguments must be the same as the number, type, and order of dummy arguments in the procedure referenced.

DUMMY ARGUMENTS

Statement functions, function subprograms, and subroutine subprograms use *dummy arguments* to indicate the types of actual arguments and whether each is a single value, an array of values, or a procedure. Statement function dummy arguments are limited to single 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.

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.

ACTUAL ARGUMENTS

Actual arguments specify the entities that are to be associated with the dummy arguments of a referenced subroutine or function. An actual argument must not be the name of a statement function in the referencing program unit. Actual arguments can be constants and expressions involving operators if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

The type of each actual argument must agree with the type of its associated dummy argument except when the actual argument is a subprogram name or alternate return.

ASSOCIATION OF DUMMY AND ACTUAL ARGUMENTS

Upon execution of a function or subroutine reference, an association is established between the corresponding actual and dummy arguments. The first actual argument becomes associated with the first dummy argument, the second actual argument becomes associated with the second dummy argument, etc.

All appearances of a dummy argument within a function or subroutine become associated with the corresponding actual argument when a reference to that function or subroutine is executed.

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 procedure.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. The subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

If an actual argument is a character substring name, its substring expressions are evaluated immediately preceding argument association. Substring expression values remain constant as long as argument association continues.

If an actual argument is an external procedure name, the procedure must be available at the time a reference to it is executed.

If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension declarator, the actual argument must be defined with an integer value at the time the procedure is referenced.

A dummy argument is undefined if it is not currently associated with an actual argument. 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.

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.

If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be less than or equal to the length of the actual argument. 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.

If a dummy argument of type character is an array name, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array can be different from the length of an array element in an associated actual argument array, array element, or array element substring, but the dummy argument array must not extend beyond the end of the associated actual argument array.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, the actual argument length is the sum of the lengths of the operands.

VARIABLES AS DUMMY ARGUMENTS

A dummy argument that is a variable can be associated with an actual argument that is a variable, array element, substring, or expression.

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 one of the following items.

- A constant
- The symbolic name of a constant
- A function reference
- An expression involving operators
- An expression enclosed in parentheses

ARRAYS AS DUMMY ARGUMENTS

Within a program unit, the array declarator given for an array provides all array declarator information required for execution of the program unit. The number and size of dimensions in an actual array declarator can be different from the number and size of the dimensions in an associated dummy array declarator.

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

If the actual argument is a noncharacter array name, the size of the dummy argument array must not exceed the size of the actual argument array. Furthermore, actual argument array elements and dummy argument array elements become associated when their subscript values match.

If the actual argument is a noncharacter array element name, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. 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 $p+q-1$.

PROCEDURES AS DUMMY ARGUMENTS

A dummy argument that is a procedure can be associated only with an actual argument that is a procedure.

If a dummy argument is used as a function, the associated actual argument must be an intrinsic function or an external function. A dummy argument that becomes associated with an intrinsic function never has automatic typing property, even if the dummy argument name is the same as the intrinsic function name. Therefore, the type of the dummy argument must agree with the type of the result of all specific actual arguments that become associated with the dummy argument. If a dummy argument name is used as an external function and that name also appears as an intrinsic function name, the intrinsic function is not available for referencing within the subprogram.

A dummy argument that is used as a procedure name in a function reference and is associated with an intrinsic function must have arguments that agree in number and type with those specified for the intrinsic function. (See the intrinsic functions in Appendix B.)

If a dummy argument appears in a type statement and an EXTERNAL statement, the actual argument must be the name of a function.

If the 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.

RESTRICTIONS ON THE ASSOCIATION OF ENTITIES

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument can become defined during execution of that subprogram. For example, if a subroutine is headed by

```
SUBROUTINE XYZ (A,B)
```

and is referenced with

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and, therefore, with each other. This rule prohibits both A and B from becoming defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram, neither the dummy argument nor the entity in the common block can become defined within the subprogram. For example, if a subroutine containing statements

```
SUBROUTINE XYZ (A)

COMMON C
```

is referenced by a program unit that contains the statements

```
COMMON B

CALL XYZ (B)
```

the dummy argument A becomes associated with the actual argument 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.

COMMON BLOCKS

A common block provides a means of communication between external procedures or between a main program and an external procedure. The variables and arrays in a common block can be defined and referenced in all subprograms that contain a declaration of that common block.

Because association is by storage sequence instead of by name, the names and types of variables and arrays can be different in different subprograms. A reference to data in a common block is proper if the data is defined and the same type as the type of the name used to reference the data. However, an integer variable assigned an executable statement label must not be referenced in any program unit other than the one where it was assigned.

The only difference in data type permitted between that defined and that referenced is that either part of a complex data can be referenced as a real data.

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.

Common blocks can also reduce the total number of storage units required for an executable program by causing two or more subprograms to share some of the same storage units. This sharing of storage is permitted if the rules for defining the referencing data are not violated. However, if any entity in a common block is of type character, all entities in the block must be of type character. Furthermore, if a common block definition in one subprogram is of type character, the common block definition in all subprograms must be of type character.

EXTENDED MEMORY COMMON BLOCKS

CFT allows common blocks to contain more than 4 million words of memory by using the extended memory addressing (EMA) characteristic (see the CPU parameter on the CFT control statement in part 3, section 1). When the EMA 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.

When a subprogram is compiled, a fatal error message is issued if very large local arrays cause the code and data storage area to exceed 4 million words. A fatal error message is also issued if any common block has more than 4 million words declared in it. Fatal error messages are not issued when all very large arrays are moved into common blocks and the EMA characteristic is used.

TASK COMMON BLOCKS

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

```
TASK COMMON/name/list
```

where *name* is the task common block name, and

list is the variable list declared.

The variables in *list* cannot be saved, preset with data, or used in the NAMELIST I/O statement. With these exceptions, the variables can be used like the other variables declared in COMMON.

Stack allocation must be used with task common blocks (see the CFT control statement in part 3, section 1). 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.

PART 2

CFT STATEMENTS

FORTRAN STATEMENTS 1

A FORTRAN statement is a sequence of syntactic items beginning, in many cases, with a keyword. The FORTRAN statement describes either the form of data and program elements or the actions to be taken by the program. A statement label can precede a statement, but is not a part of the statement itself.

The type of a statement is indicated by its keyword or by its form. The total number of characters expressing a statement is limited to 1,320, including blank characters. Aside from character-count limitation, leading, trailing, and, except within character constants, embedded blank characters do not affect statement interpretation.

This section describes ANSI FORTRAN statements and some CFT extensions to these statements, as well as some additional CFT statements. These extensions enhance the capability of the Cray FORTRAN language.

Appendix E describes several non-standard, outmoded statements and features which, although supported by CFT, can be conveniently replaced with standard features.

DATA SPECIFICATION 2

Data specification statements are statements supplying characteristics and values of data used in the execution of a program. Data specification statements are not executable; that is, they form no execution sequence. Therefore, statement labels associated with them cannot be referenced to control the execution sequence. Data specification statements usually appear (and some must appear) before any executable statements in a program.

Types of data specification statements are as follows.

- Declaration and initialization
- Type
- Association

DECLARATION AND INITIALIZATION

Declaration and initialization statements provide values and locations and establish arrays.

PARAMETER STATEMENT

A PARAMETER statement assigns a symbolic name to a constant.

The format of a PARAMETER statement is

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

where *p* is a symbolic name, and

e is an expression containing constants and symbolic names of other constants.

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* of type integer, real, double precision, or complex is followed only by an arithmetic expression *e* containing arithmetic constants or the names of arithmetic constants previously defined in the same or an earlier 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, a default length of one is assumed. The length cannot be changed by subsequent statements. If the length of (*) is specified, the parameter length is the length of the actual character string.

The evaluation of arithmetic expressions in a PARAMETER statement provides results agreeing in type with the corresponding symbolic names. A symbolic name *p* of type logical is followed only by a logical constant expression. Similarly, a symbolic name *p* of type character is followed only by a character constant expression. A symbolic name of a constant is assigned a value only once in a program unit. Constants named in a PARAMETER statement can be referenced in a subsequent statement in the same program unit except in a FORMAT statement. A symbolic name of a constant cannot be used in a format specification or to form part of another constant.

Examples:

```
IMPLICIT LOGICAL(A-B)

PARAMETER (PI=3.1415926, C=1.86E5)

PARAMETER (JOULE=10000000,KELVIN=-273)

PARAMETER (BOOLEAN=.TRUE.,ABOOLEAN=.FALSE.,TWOPI=2*PI)
```

DIMENSION STATEMENT

The DIMENSION statement specifies the symbolic names and dimension specifications of arrays.

The format of a DIMENSION statement is

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

where each *a*(*d*) is an array declarator.

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. The declaration for a variable used in adjustable dimensions must precede the adjustable dimension declaration.

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.

POINTER STATEMENT (CFT EXTENSION)

The POINTER statement provides a base address for a corresponding variable or array.

The format of a POINTER statement is

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

where *p* is a pointer to the corresponding *a*. *p* contains the word address of the location of *a*.

No storage is assigned for *a*; a reference to *a* is performed by using the contents of *p* as a base address for *a*.

a 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 common or to dummy arguments.

a cannot be a dummy argument or type character or appear in a COMMON, EQUIVALENCE, or DATA statement.

The pointer, *p*, has an implied type of integer and must be a simple variable. It can appear in a common list or be a dummy argument in a subprogram. A maximum of 312 pointers can be defined in any program unit. *p* can be set with an LOC function reference or as an absolute address, as in the following example.

```
COMMON POOL (100000)
INTEGER JCB (128)
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 equivalenced to POOL and possibly to each other, depending on the subscript usage. Similarly, WORD64 is the same as JCB(64). However, CFT makes no checks for possible equivalence overlap. Each *a* is assumed to be a distinct entity.

Any change to the value of a *p* causes all subsequent references to the corresponding *a* 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
```

DATA STATEMENT

A DATA statement provides initial values for variables, arrays, and array elements. A DATA statement can appear in a program unit following a specification statement. Only those entities named in DATA statements become defined before executable program execution. All other entities are undefined at this time.

Entities appearing in DATA statements are assigned to static storage.

The ANSI FORTRAN Standard does not specify storage allocation methods.

The format of a DATA statement is

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

where *nlist* is a list of variable names, array names, array element names, substring names, and implied-DO lists separated by commas, and

clist is a list of the form

[r]c[,[r*]c]...*

where *c* is a constant or the symbolic name of a constant, and

r is a 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*.

The *i*th entity in *nlist* becomes defined with the *i*th value from *clist*.

An implied-DO list in a DATA statement has the format

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

where *dlist* is a list of array element names and implied-DO lists separated by commas,

i is the name of an integer variable called the *implied-DO variable*, and

e_1, e_2, and e_3 are integer expressions containing integer constants, the names of integer constants, and implied-DO variables of other implied-DO lists containing the implied-DO list within their ranges. If omitted, e_3 is assumed specified as 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 the same as a DO-loop except 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 the expression may contain implied-DO variables of implied-DO lists having the subscript expressions in their ranges.

In the following example, the first ten values of array A are set to 1.

```
DIMENSION A(25)
DATA (A(I),I=1,10)/10*1/
```

DATA statement restrictions

Names of constants, dummy arguments, functions, and entities in blank common (including entities associated with an entity in blank common) must not appear in *nlist*. Names of entities in a named common block can appear in *nlist*.

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.

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. 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 *nlist* entity corresponding to a character constant must be of type character. If the character entity length in the list *nlist* is greater than its corresponding character constant length, the additional rightmost characters in the entity are initially defined with blank characters. If the character entity length in the list *nlist* is less than its corresponding character constant length, the additional rightmost characters in the constant are ignored.

Any variable or array element can be initially defined except for the following.

- An entity that is a dummy argument
- An entity in blank common, which includes an entity associated with an entity in blank common

- A variable in a function subprogram whose name is also the name of the function subprogram

Subscript expressions in the list *nlist* must be integer constant expressions except for implied-DO variables. Substring expressions in the list *nlist* must also be integer constant expressions.

Any declaratives affecting the variable or array names in *nlist* must precede the DATA statement.

The ANSI FORTRAN Standard does not permit a DATA statement to initialize entities in named common blocks except in block data subprograms.

Examples:

```
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/0/K/2000/

PARAMETER (NEG=-6)

INTEGER NB (10)

DATA NB/-3,7*-4,2*NEG/
```

TYPE STATEMENTS

A type statement either overrides or confirms implicit typing and can specify dimension information.

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 appearing in a type statement conflicts with that function's type as specified in Appendix B, the conflicting type statement is ignored but a warning message is issued.

INTEGER, REAL, DOUBLE PRECISION, COMPLEX, AND LOGICAL TYPE STATEMENTS

The format of type statements is

```
type v[,v]...
```

where *type* specifies type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and

v is the symbolic name of a constant, variable name, array name, function name, dummy procedure name, or array declarator.

The space between DOUBLE and PRECISION is optional.

(A special form of integer typing is discussed in part 3, section 1, Compiler Directives.)

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)
```

See Appendix E for extensions of the type declaration statements.

CHARACTER TYPE STATEMENT

The format of a CHARACTER type statement is

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

where *len* is the length specification (number of characters) for an entity and

nam is a symbolic name of a constant, variable name, function name, dummy procedure name, 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* must be less than 16,384. If the entity is an external function, a dummy argument, or a character constant, *len* can also be specified as an asterisk enclosed in parentheses (for example, CHARACTER*(*)).

If the entity is an external function and the value of *len* is specified as (*), the function name must appear in a FUNCTION or an ENTRY statement in the same subprogram. The value of *len* is the length specified in the referencing program unit.

If the entity is a dummy argument and the value of *len* is specified as (*), the dummy argument assumes the length of the associated actual argument.

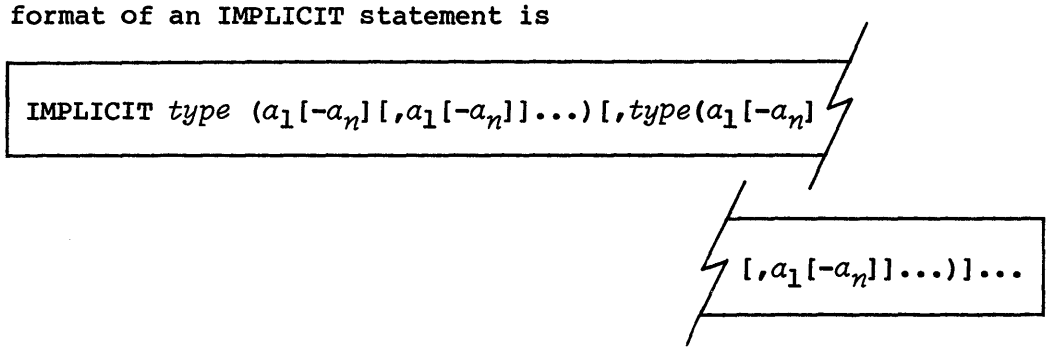
The ANSI FORTRAN Standard does not specify a maximum character length.

If the entity is a character constant with a symbolic name and the value of *len* is specified as (*), the constant assumes the length of its corresponding constant expression defined in a PARAMETER statement.

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.

The format of an IMPLICIT statement is



where *type* is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER[**len*], or LOGICAL to specify the desired data type;

a is 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-a_n) has the same effect as writing a list of the single letters (a_1,a_2,...a_n) where a_1 precedes a_n in this alphabetically ordered sequence; and

len is the length of the character entities. *len* can be an unsigned, nonzero, positive integer constant or expression with a value less than 16,384.

An IMPLICIT statement specifies a type for all constant, variable, array, and function (except intrinsic function) names 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 INTEGER(A,B,F-K),REAL(M-W,Z)

IMPLICIT LOGICAL(L)

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

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 format of an IMPLICIT NONE statement is

```
IMPLICIT NONE
```

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

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, therefore, need not be explicitly typed as such.

ASSOCIATION STATEMENTS

Association statements specify the relationship of entities to storage units within the same program or among two or more programs.

EQUIVALENCE STATEMENT

An EQUIVALENCE statement specifies the sharing of one or more storage units by two or more entities in a single program unit. This causes the association of those entities.

If associated entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. 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.

Associated entities are assigned to the same type of storage, static or stack storage. Associated entities are assigned to static storage unless the ALLOC=STACK option is specified on the CFT control statement (see part 3, section 1) and the entities have not been previously assigned to static storage (for example, with a DATA statement).

The ANSI FORTRAN Standard does not specify storage allocation methods.

The format of an EQUIVALENCE statement is

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

where *nlist* is a list of two or more variable names, array element names, character substring names, and array names, separated by commas.

Names of dummy arguments of a subprogram cannot appear in *nlist*. A variable name that is also a function name cannot appear in *nlist*.

Each subscript expression or substring expression in *nlist* must be an integer constant expression.

Equivalence association

An EQUIVALENCE statement specifies that the storage sequence of each entity in a list *nlist* shares the same first storage unit. This causes the association of all entities in the list and can also cause indirect association of other entities.

Array names and array element names

If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must equal the number of dimensions in the array declarator for the array.

The use of an array name in an EQUIVALENCE statement has the same effect as using the name of the first array element.

Restrictions on EQUIVALENCE statements

An EQUIVALENCE statement must not specify the same storage unit to occur more than once in a storage sequence. For example,

```
DIMENSION A(2)

EQUIVALENCE (A(1) ,B) , (A(2),B)
```

is prohibited because it would specify 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))
```

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)
```

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. For example,

```
COMMON /X/A

REAL B(2)

EQUIVALENCE (A,B(2))
```

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 equivalence association.

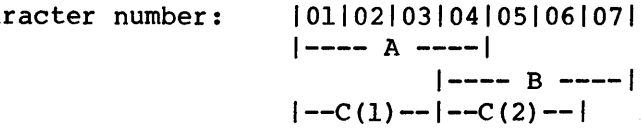
Example:

The appearance of

```
CHARACTER A*4,B*4,C(2)*3

EQUIVALENCE (A,C(1)),(B,C(2))
```

in a program unit associates A with C(2) as shown in the following illustration.



COMMON STATEMENT

The COMMON statement associates entities in different program units. This allows different program units to share storage units and define and reference the same data.

The format of a COMMON statement is

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

where *cb* is a common block name, and

nlist is a list of variable names, array names, and array declarators separated by commas. Names of dummy arguments of a subprogram cannot appear in the list.

In each COMMON statement, the entities occurring in *nlist* following a block name *cb* are declared to be in common block *cb*. The blank (unnamed) common block is specified when a *cb* does not appear between slashes. If the first *cb* is omitted, its enclosing slashes are optional and all entities in *nlist* are specified to be in blank common.

A *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 an 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 another procedure, the entities in that procedure must be character entities.

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 in *nlist* for the common block. The order of the storage sequence is determined by the order of the appearance of *nlist* in the program unit.

- The storage sequence is extended to include all storage units of any storage sequence associated with it by EQUIVALENCE statement association. 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.

Size of a common block

The size of a common block is the size of its common block storage sequence, including any extensions of the sequence resulting from EQUIVALENCE statement association.

In an executable program, the size of a named common block is established during compilation of the first program unit specifying its name. The size cannot be exceeded in specifying the same named common block in subsequent program units, but can be the same or less. Blank common blocks in an executable program are not required to be the same size and can increase, decrease or remain the same as each program unit is compiled.

The ANSI FORTRAN Standard does not provide for variable size for named common blocks.

Common association

In an executable program, 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 with different program units.

Differences between named common and blank common

A blank common block has the same properties as a named common block except entities in blank common blocks cannot be initially defined by DATA statements.

Restrictions on COMMON and EQUIVALENCE statements

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit. 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.

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.

The format of an INTRINSIC statement is

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

where *fun* is an intrinsic function name.

The appearance of a name in an INTRINSIC statement declares the name is 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 the program unit.

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

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

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.

SAVE STATEMENT

A SAVE statement retains the definition status of an entity after the execution of a RETURN or END statement in a subprogram. 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.

The format of a SAVE statement is

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

where *a* is a named common block name preceded and followed by a slash; a variable name; or an array name.

The names of dummy arguments, pointers, or procedures must not be specified in a SAVE statement. Variables and arrays in 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 cannot appear more than once in the SAVE statements of a program unit.

A SAVE statement is optional in the main program and has no effect on the main program.

See the CFT control statement option BTREG in part 3, section 1 for more information on SAVE statement allocation.

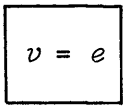
ASSIGNMENT STATEMENTS 3

Assignment statements define variables and array elements during program execution. Categories of assignment statements are as follows.

- Arithmetic
- Logical
- Character
- ASSIGN (statement label)

ARITHMETIC ASSIGNMENT STATEMENT

The format of an arithmetic assignment statement is



where *v* is the name of a variable or array element of type integer, real, double-precision, or complex, and

e is an arithmetic expression.

Execution of an arithmetic assignment statement causes the evaluation of the expression *e*, conversion of *e* to the type of *v* (if required), and definition of *v* with the resulting value. Table 3-4 in part 1, section 3 relates such conversions to the data types of arithmetic operands, expressions, and evaluations.

Examples:

The statement ...	Assigns to *v* ...
`L = 12`	Integer variable
`C = (0.8,16.5) - (16.32,-6.1)`	Complex variable
`X = -B +(B**2-4*A*C)**0.5`	Real variable
`A= B + L`	Real variable
`ROOT = SQRT(65536.0)`	Real variable
`ARRAY(6,2,1)=0`	Real array element
`MATRIX(I,J,K)=MATRIX(I,J,K)+1`	Integer array element

LOGICAL ASSIGNMENT STATEMENT

The format of a logical assignment statement is

```
v = e
```

where *v* is the name of a logical variable or array element, and

e is a logical expression.

Execution of a logical assignment statement causes the evaluation of the expression *e* and the definition of *v* with the value of *e*.

Examples:

All variable and array element names are assumed to be of type logical except for 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
```

CHARACTER ASSIGNMENT STATEMENT

The format of a character assignment statement is

```
v = e
```

where *v* is the name of a variable, array element, or substring of type character, and

e is a character expression.

Execution of a character assignment statement causes the evaluation of the expression *e* and the definition of *v* with the value of *e*. *e* is either truncated or padded with blanks on the right, as necessary, to match the length of *v*. No character positions defined in *v* can be referenced in *e*.

Example:

The following sequence

```
CHARACTER  DATE*12, MONTH*9, DAY*2, YEAR*4
MONTH = 'OCTOBER'
DAY = '3'
YEAR = '1982'
DATE = MONTH(1:3)//' '//DAY//', '//YEAR
PRINT *,DATE
```

produces the printed result

```
OCT 3 , 1982.
```

ASSIGN STATEMENT

The format of an ASSIGN statement is

```
ASSIGN s TO i
```

where *s* is a statement label, and

i is a integer variable name.

An ASSIGN statement assigns the statement label *s* to the integer variable *i*. *s* must be the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

Execution of an ASSIGN statement is the only way to define a variable with a statement label.

A variable defined with an executable statement label can be referenced only in an assigned GO TO statement. A variable defined with a FORMAT statement label can be referenced as a format identifier in an I/O statement. While so defined, the variable *i* cannot be referenced for any other purpose. However, the variable *i* can be redefined with another statement label or an integer value. In the latter case, it can be used anywhere an integer variable is used.

Example:

```
ASSIGN 910 TO JUMPTO
```

PROGRAM CONTROL STATEMENTS 4

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 GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- Logical IF
- Conditional block statements
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL (Described in section 7)
- RETURN (Described in section 7)

UNCONDITIONAL GO TO STATEMENT

The format of an unconditional GO TO statement is

```
GO TO s
```

where *s* is the statement label of an executable statement in the same program unit.

The space between GO and TO is optional.

Execution of an unconditional GO TO statement causes a transfer of control to the statement identified by the statement label.

Example:

```
GO TO 910
```

COMPUTED GO TO STATEMENT

The format of a computed GO TO statement is

```
GO TO (s[,s]...) [,]e
```

where *e* is an integer expression, and

s is the statement label of an executable statement that appears in the same program unit as the computed GO TO statement. A given statement label can appear more than once in a computed GO TO statement.

The space between GO and TO is optional.

Execution of a computed GO TO 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 $1 \leq i \leq n$. 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 non-integer result, it is converted to integer as if *i*=*e* had been executed.

Examples:

```
GO TO (2,4,8,16)A    (The value of A is truncated, if necessary, to
                      produce an integer value.)

GO TO (0031,59,728)IX
```

```
GO TO (0031,59,728)MSIZE/2

GO TO (6,3,6,6,7,2,7),NBRANCH
```

ASSIGNED GO TO STATEMENT

The format of an assigned GO TO statement is

```
GO TO i[[,] (s[,s]...)]
```

where *i* is an integer variable name, and

s is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. A given statement label can appear more than once in this statement.

The space between GO and TO is optional.

At the time of execution of an assigned GO TO statement, the variable *i* must be defined with the value of a statement label appearing in the same program unit. The variable can be defined with a statement label value by an ASSIGN statement in the same program unit as the assigned GO TO statement. Execution of the assigned GO TO statement causes a transfer of control to the statement identified by that statement label.

CFT does not use the optional statement list.

The ANSI FORTRAN Standard specifies that if the optional list is present, *i* must have been assigned a statement label from the list.

Examples:

```
ASSIGN 76 TO LAB
  .
  .
  .
GO TO LAB

ASSIGN 999 TO KFIN
  .
  .
  .
GO TO KFIN (997,997,999)
```

```
ASSIGN 1 TO JAIL
  .
  .
  .
GO TO JAIL,(1,2,3,4,5)
```

ARITHMETIC IF STATEMENT

The format of the arithmetic IF statement is

```
IF (e) s1,s2,s3
```

where *e* is an integer, real, or double-precision expression, and

*s*1,*s*2, and *s*3 are 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.

Executing an arithmetic IF statement evaluates 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.

Examples:

```
IF (VTEST)  20,21,20

IF (B**2-4*A*C) 70,80,90
```

LOGICAL IF STATEMENT

The format of a logical IF statement is

```
IF (e) st
```

where *e* is a logical expression, and

st is any executable statement other than a DO, END, block IF, ELSE IF, ELSE, END IF, or another logical IF statement.

Executing a logical IF statement evaluates 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 if a CONTINUE statement were executed. The execution of a function reference in the expression *e* may affect entities in the statement *st*.

Examples:

```
IF(K) K=.NOT.K

IF (A.EQ.B) GO TO 100
```

CONDITIONAL BLOCK STATEMENTS

Conditional block statements delimit groups of executable statements called blocks. They control the execution sequence of the statements in a block. Following is a list of the conditional block statements.

- Block IF
- END IF
- ELSE IF
- 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 must always be 0 or positive; the IF-level of the END statement of each program unit must always be zero.

IF-BLOCK

An IF-block is a group of executable statements preceded by a block IF statement and followed by another conditional block statement (END IF, ELSE IF, or ELSE) of the same IF-level. An IF-block can be empty.

BLOCK IF STATEMENT

The format of the block IF statement is

```
IF (e) THEN
```

where *e* is a logical expression.

Executing the block IF statement evaluates 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 END IF, ELSE IF, or ELSE statement of the same IF-level. The block IF statement must always have a corresponding END IF statement of the same IF-level.

If a block IF statement appears within the range of a DO-loop, the entire block must appear within the range.

Control cannot be transferred into an IF-block from outside the IF-block.

END IF STATEMENT

The format of the END IF statement follows.

```
END IF
```

The END IF statement indicates the end of an IF-level and must always have a corresponding block IF statement of the same IF-level.

The space between END and IF is optional.

ELSE IF-BLOCK

An ELSE IF-block is a group of executable statements with an ELSE IF statement preceding the group and a conditional block statement (END IF, ELSE IF, or ELSE) of the same IF-level following the group. An ELSE IF-block can be empty. The IF-level of the ELSE IF-block must be greater than or equal to 1.

ELSE IF STATEMENT

The format of the ELSE IF statement is

```
ELSE IF (e) THEN
```

where *e* is a logical expression.

The ELSE IF statement is executed if none of the preceding blocks have been executed. Execution of the ELSE IF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution sequence continues with the first statement of the ELSE IF-block. If the value of *e* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. Statement labels on ELSE IF statements are ignored.

The space between ELSE and IF is optional.

Control cannot be transferred into an ELSE IF-block from outside the ELSE IF-block.

ELSE-BLOCK

An ELSE-block is a group of executable statements with an ELSE statement preceding the group and an END IF statement of the same IF-level following the group. No other conditional block statement at the same level can appear after the ELSE statement or before the END IF statement. ELSE-blocks can be empty. The IF-level of the ELSE-block must be greater than or equal to 1. Statements in the ELSE-block are executed if none of the preceding blocks were executed.

ELSE STATEMENT

The format of the ELSE statement follows.

```
ELSE
```

The ELSE statement introduces an ELSE-block. Statement labels on ELSE statements are ignored.

CONDITIONAL BLOCK STATEMENT EXECUTION

A group of blocks must begin with a block IF statement and end with an END IF 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 ELSE IF and ELSE statements are not required to accompany block IF statements. A block begins with a block IF, an ELSE IF, or an ELSE statement and continues until an END IF 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. (See figure 4-1 for an illustration of blocks and levels.) 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 END IF 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 END IF statement is encountered.

If an END IF statement is encountered, the IF-level is decremented by 1 and all following statements retain that IF-level number until a block IF or END IF is encountered.

DO STATEMENT

A DO statement specifies necessary information to control the repeated execution of a set of statements. A *DO-loop* consists of a DO statement, the set of statements to be executed repeatedly, including a labeled terminal statement.

The format of a DO statement is

DO s[,]i = e_1,e_2[,e_3]

where s is the statement label of an executable statement, called the terminal statement;

i is the name of an integer, real, or double-precision variable, called the DO variable; and

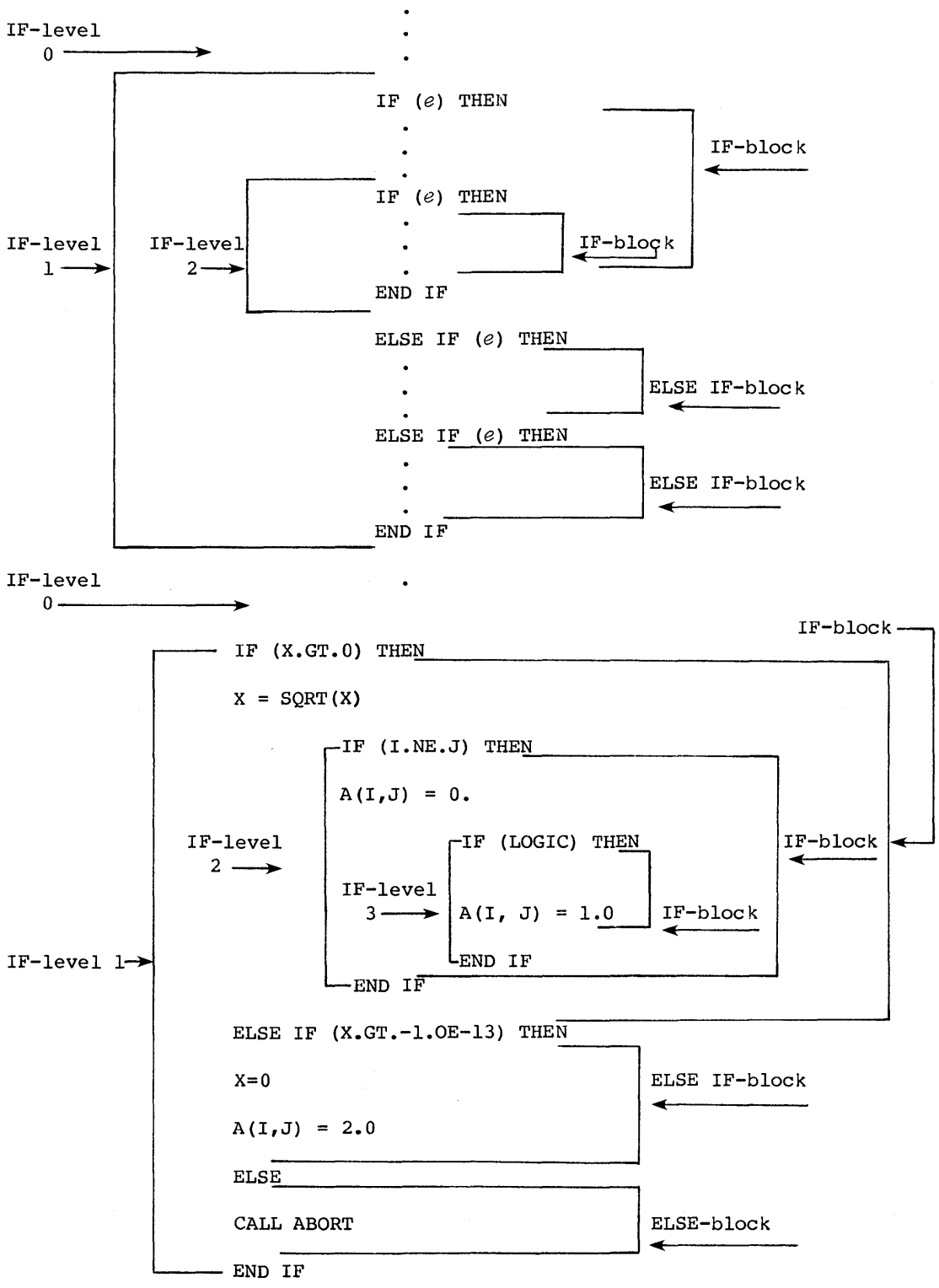


Figure 4-1. IF-levels and blocks

e_1, e_2, and e_3
are 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.

TERMINAL STATEMENT

The *terminal statement* is an executable statement ending the DO-loop. The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, 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 can contain any executable statement except a DO, conditional block, END, or another logical IF statement.

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.

RANGE OF A DO-LOOP

The range of a DO-loop consists of all executable statements, beginning with 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 it 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 also appear within the range of that DO-loop.

The following example contains constructs classified by the ANSI FORTRAN standard as illegally jumping into the range of a DO-loop. This type of construct should be avoided.

Example:

```
      DO 10 I=1,20
      A(I)=0.
      IF(I.GT.10)GO TO 10
      DO 10 J=1,100
      B(J,I)=0.
10    CONTINUE
```

ACTIVE AND INACTIVE DO-LOOPS

A DO-loop is either active or inactive. A DO-loop is initially inactive and becomes active only when its DO statement is executed.

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.
- It is in the range of another DO-loop that becomes inactive.
- It is in the range of another DO-loop having an executed DO statement.

When a DO-loop becomes inactive, the DO variable retains its last defined value unless it became undefined due to earlier action.

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 $2^{23}-1$ (8,388,607) in absolute value or undetected bad code may be produced.

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 expression

   $$(m_2-m_1+m_3)/m_3$$

   or as 0 in the event that

   $$m_1>m_2, \text{ and } m_3>0 \text{ or}$$
   $$m_1<m_2, \text{ and } m_3<0.$$

   $m_3=0$ 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} (8,388,608). Once the iteration count is established, the DO variable and 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 completion of DO statement execution, loop control processing begins.

The ANSI FORTRAN Standard does not permit the DO variable to be redefined during execution of the DO-loop range.

The ANSI FORTRAN Standard does not specify a maximum iteration count.

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 CFT control statement overrides this feature and causes execution of all DO-loops 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 below.

EXECUTION OF THE RANGE

Statements in the range of a DO-loop are executed until the terminal statement is reached.

TERMINAL STATEMENT EXECUTION

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, as described below.

INCREMENTATION PROCESSING

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.

The value of the DO variable at termination is not defined if the DO variable was redefined in the range of the DO loop.

TRANSFER INTO THE RANGE OF A DO-LOOP

Control must not transfer into the range of an inactive DO-loop.

Examples:

```
    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)
         .
         .
         .
```

```
    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
      .
      .
      .
```

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

In the last example, I=11, J=10, K=6, L=5, and M=50 after the last statement is executed for the last time.

CONTINUE STATEMENT

The format of a CONTINUE statement follows.

```
CONTINUE
```

Execution of a CONTINUE statement has no effect.

A CONTINUE statement 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.

Examples:

```
        DIMENSION ARRAY6(16)
        DO 22,I=16,1,-1
        IF(ARRAY6(I).NE.0) ARRAY6(I)=1.0/ARRAY6(I)
   22   CONTINUE
```

STOP STATEMENT

The format of a STOP statement is

```
STOP [id]
```

where *id* is an unsigned integer constant of up to eight digits, a character constant of up to eight characters, or the name of a character variable, array element, or function containing (or providing) eight characters.

The ANSI FORTRAN Standard limits noncharacter *id* to five 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.

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.

PAUSE STATEMENT

The format of a PAUSE statement is

```
PAUSE [id]
```

where *id* is an unsigned integer constant of up to eight digits, a character constant of up to eight characters enclosed in parentheses, or the name of a character variable, array element, or function containing or providing eight characters.

The ANSI FORTRAN Standard limits noncharacter *id* to five 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.

A PAUSE statement suspends or terminates a main program, subroutine subprogram, or function subprogram. 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.

END STATEMENT

The format of an END statement follows.

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 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.

INPUT/OUTPUT STATEMENTS 5

Input statements transfer data from datasets to the memory section of the CPU. This process is called *reading*. *Output statements* transfer data from memory to datasets. This process is called *writing*. *Editing* of the data can be performed by using format identifiers.

This section describes input/output operations to COS blocked datasets. Limited input/output operations provided for unblocked datasets are described in Appendix H. Nonstandard random access input/output operations are described in Appendix E.

INPUT/OUTPUT RECORDS

A record is a sequence of values or characters. For example, a punched card is usually considered 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.

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 input/output statements, or prepared by means other than FORTRAN.

Unformatted and buffered input/output 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 ANSI FORTRAN Standard allows reading and writing of formatted records only by formatted I/O.

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 takes ((*len*-1)/8)+1 words.

Unformatted records can be read or written by unformatted and buffered input/output statements.

The ANSI FORTRAN Standard does not allow reading and writing of unformatted records with formatted I/O.

END-OF-FILE (ENDFILE) RECORDS

An endfile record is written by an ENDFILE statement, must occur only as the last record of a file, and has no length property.

END-OF-DATA RECORDS

An end-of-data (EOD) occurs on the last record of a dataset. It cannot be explicitly written by a FORTRAN program.

INPUT/OUTPUT FILES

A *file* is a sequence of records. A file *exists* for an executable program when the file is identified and/or referenced by a name. A file can be present and not exist for an executable program at a specific time. A file can also exist with no records.

Creating a file brings a file into existence. *Deleting* a file terminates its existence.

RECORD AND FILE POSITIONS

Because records and files exist as elements in sequences, the position of a record or a file can be described by its position in a sequence. Certain circumstances can cause this position to become indeterminate.

In a sequence, the *initial point* is the position just before the first element. The *terminal point* is the position just after the last element.

If a sequence is positioned at a point within an element, that element is the *current element*; otherwise, no current element exists.

A *preceding element* is that element preceding the current element or terminal point. No preceding element exists for the initial point of a sequence or for the terminal point if the sequence is empty.

The *next element* of a sequence immediately follows the current element. No next element exists for the terminal point of a sequence or for the initial point if the sequence is empty.

A file can contain formatted and unformatted records and is terminated with an endfile record.

The ANSI FORTRAN Standard does not provide for the mixing of formatted and unformatted records in a file.

DATASETS

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 under control of the executable program. However, datasets and units can be preassociated before program execution. Datasets are described in the CRAY-OS Version 1 Reference Manual, publication SR-0011.

The ANSI FORTRAN Standard does not provide for datasets or other multiple file entities.

INTERNAL RECORDS AND FILES

Internal records and *internal files* are analogous to records and files except an internal file identifier is used in place of an external unit identifier. Internal files provide a way to transfer and convert data within internal storage.

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 variable, array element, or substring, respectively. 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, that is, the variable, array element, or substring, is 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 undefined 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.

An internal file has the following restrictions.

- Reading and writing records is done only by sequential access formatted input/output statements not specifying list-directed formatting.

- An auxiliary input/output statement must not specify an internal file.

SEQUENTIAL ACCESS OPERATIONS

Sequential access operations are based on the sequential storage of records within files and files within datasets. The order of the records is the order in which they are written. If direct access is also allowed, the order of 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.

The last record of a file must be an endfile record. The records of a file must not be read or written by direct access input/output statements while the dataset is connected for sequential access.

DIRECT ACCESS OPERATIONS

In direct access operations, records can be read or written in any order. The order of record numbers specifies the order of the records.

All records of the dataset have the same length and each record of a dataset has a unique record number. The record number is a positive integer 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. Direct access input/output statements must be used for reading and writing while the dataset is connected for direct access.

Multifile direct access datasets are not allowed.

See Appendix E for random access extensions.

DATASET POSITION BEFORE DATA TRANSFER

Dataset position depends on the method of access, sequential access or direct access.

Sequential access

When an input operation is performed on a dataset, the dataset is positioned at the beginning of the next record, becoming the current record. When an output operation is performed on a dataset, a new record is created, becoming the last record of the dataset.

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.

Direct access

The dataset is positioned at the beginning of the record specified by the record specifier, becoming the current record.

UNITS

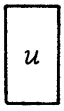
A *unit* is a means of referring to a dataset. At any given time, a set of units *exists* for an executable program. All input/output statements can refer to existing units.

IDENTIFIERS

Identifiers assign names to units, internal files, datasets, and formats.

UNIT IDENTIFIERS

The format of a *unit identifier* is



where *u* is an external unit identifier (type integer) or an internal file identifier (type character).

An *external unit identifier* is used to refer to an external dataset and is an integer constant or expression in the range 0 through 102, or the character *. The values 100, 101, and 102 refer to datasets $IN, $OUT, and $PUNCH, 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; in a WRITE statement, it refers to $OUT. The default for units 5 and 6 are $IN and $OUT, respectively. These assignments can be changed. (See the CRAY-OS Version 1 Reference Manual, publication SR-0011 for details on dataset definition and control.)

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 specifier, the unit specifier must be the first item in a list of specifiers.

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.

DATASET IDENTIFIERS

The format of a *dataset identifier* follows.

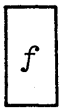
A dataset identifier is a character constant, an integer variable, or an integer array element containing Hollerith data of not more than seven characters whose name identifies the dataset.

External dataset identifiers of type character can only be used in OPEN and INQUIRE statements. Using character type variables as unit specifiers in READ/WRITE statements implies I/O operations on internal files.

The ANSI FORTRAN Standard does not provide for dataset identifiers.

FORMAT IDENTIFIERS

The format of a *format identifier* follows.



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

READ, WRITE, AND PRINT STATEMENTS

The READ statement is the data transfer input statement. WRITE and PRINT statements are data transfer output statements. These statements have the following formats.

```
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.

where *cilist* is a control information list that includes a reference to the source or destination of the data to be transferred and an optional format identifier for editing processes,

f is a format identifier, and

iolist is an input/output list specifying the data to be transferred.

The format of a *control information list* (*cilist*) is

[UNIT=] $\left\{ \begin{matrix} u \\ din \end{matrix} \right\}$ [,[FMT=]*f*] [,END=*sn*] [,REC=*rn*] [,ERR=*s*] [,IOSTAT=*ios*]

where [UNIT=]*u* is the unit specifier and [UNIT=]*din* is the dataset specifier. The control information list must contain a unit specifier or a dataset specifier, but not both. If the UNIT= keyword is omitted, *u* and *din* are positional parameters and must appear first.

The ANSI FORTRAN Standard does not provide for the [UNIT=]*din* form.

[FMT=]*f* is the format specifier. This parameter must be present for formatted input/output statements. If *f* is an asterisk, the statement is list-directed and a record specifier cannot be present. If the optional UNIT= keyword is specified with the unit or dataset 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 *din*.

END=*sn* indicates an end-of-file specifier. *sn* is the number of the statement where execution continues after an EOF on a READ statement has been encountered. An end-of-file specifier must not appear in a WRITE statement or in the same control information list as a record specifier.

REC=*rn* indicates a record specifier. *rn* must be a positive integer expression with a positive value. A record specifier appears only in direct-access input/output statements. A statement containing a record specifier cannot contain an end-of-file specifier.

ERR=*s* indicates an error specifier. *s* is the statement label of the statement where control continues after encountering a recoverable error.

IOSTAT=*ios*

is a status specifier that becomes defined when an input/output statement is executed. *ios* must be an integer variable or an integer array element. Following are the specifier 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 CRAY-OS Message Manual, publication SR-0039.
<0	End of file was encountered; no error condition exists.

Examples:

```
READ(98,12345,ERR=42,END=75)...
READ(10,REC=J)...
READ(J,ARRAYF,ERR=10,END=25)...
READ *,JOE...
PRINT 22
READ(*,*)JOE
READ(10,*)JOE
READ(98,'(6E11.4)',END=75)
```

```
WRITE(17,25)...
READ(K+1)...
READ(98,ERR=37)...
READ(END=100,FMT=20,UNIT=5)...
READ(10,IOSTAT=JOE)
READ(10,IOSTAT=JOE,ERR=100,END=200)
```

INPUT/OUTPUT LISTS

An *input/output list* (*iolist*), specifies entities whose values are transferred by input/output statements. This list is composed of one or more input/output list items separated by commas. Optionally, one or more implied-DO lists can be included in the list.

An array name appearing as an input/output list item is treated as if all elements of the array were specified in the order given by array element ordering.

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

Output list items

An *output list item* must be one of the following.

- Variable name
- Array element name
- Array name
- Character substring name
- Expression other than a character expression with concatenation of an operand with a length specification of (*), unless the operand is the symbolic name of a constant

Example:

```
CHARACTER*10 C,D

DIMENSION A(10),FARRAY(3)

READ(23)X,Y

READ(23,1066)A(1),A(4),X(1),X(2)

READ(K)C,A,D(2:3)

WRITE(7)A+B
```

Implied-DO lists

The format of an implied-DO list is

$$(dlist, i = e_1, e_2[, e_3])$$

where *dlist* is an input/output list, and

i, e_1, e_2, and e_3 are as specified for the DO statement.

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 input/output 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)(BUFF(I),I=1,1000)

READ(5,1,END=50,ERR=60)((BUFFER(I,J),I=1,20),J=1,1000)
```

DATA TRANSFER

When a data transfer input/output statement is executed, the following operations are performed in the order specified.

1. Determine the direction of data transfer.
2. Identify the unit.
3. Establish the format (if specified).
4. Transfer data between the dataset or the internal file and the entities specified by the input/output list (if any).
5. The status specifier is defined (if specified).

DIRECTION OF DATA TRANSFER

Execution of a READ statement causes values to be transferred from a dataset or internal file to the entities specified by the input list, if present.

Execution of a WRITE or PRINT statement causes values to be transferred to a dataset or internal file from the entities specified by the output list and format specification (if any). The WRITE and PRINT statements are treated identically in this regard. Execution of a WRITE or PRINT statement for a non-existent dataset creates that dataset.

IDENTIFYING A UNIT

A READ statement without a unit or dataset specifier specifies the predetermined unit, preconnected to identifier 100 corresponding to the dataset $IN. PRINT statements similarly specify the preconnected identifier 101 corresponding to the dataset $OUT. Unit preconnection for PRINT statements and READ statements without a unit specifier is not under the control of the executable program.

If the dataset specified by the output statement does not exist, a dataset is created and the write proceeds normally. If the dataset specified by an input statement does not exist, an empty dataset is created and the input statement reads the end-of-data (EOD).

The ANSI FORTRAN Standard does not provide for reading or writing a non-existent file (dataset).

ESTABLISHING A FORMAT

A format identifier in a control information list identifies a format specification.

DATA TRANSFER

Data is transferred between records and entities specified by the input/output list. List items are processed in the order of their left-to-right appearance in the input/output list.

All values needed to determine entities specified by an input/output list item are determined at the beginning of the processing of that item. In the example,

```
N(1) = 3

READ (8) N(N(1))
```

a value reads into N(3). The array element item is a single input/output list item.

All values are transmitted to or from the entities specified by a list item before the processing of any succeeding list item. In the example,

```
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 input/output list item.

An input list item, or any entity associated with it, must not affect any portion of the established format specification.

Unformatted data transfer

During unformatted data transfer, data is transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the dataset 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.

Formatted data transfer

During formatted data transfer, data is transferred with editing between the entities specified by the input/output list and the dataset. The current record and possibly additional records are read or written.

On input, the record read should be a formatted or endfile record.†

The input/output 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.

† CFT allows formatted and unformatted records on the same file or dataset (non-ANSI).

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.

The transfer of formatted record information to certain devices is termed *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 5-1.

Table 5-1. 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.

ERROR AND END-OF-FILE CONDITIONS

If an error condition exists, the position of the dataset is indeterminate.

If an end-of-file (EOF) condition exists as a result of reading an endfile record, the dataset is positioned after the endfile record.

If no error condition or EOF condition exists, the dataset 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.

BACKSPACE, ENDFILE, AND REWIND STATEMENTS

The formats of the BACKSPACE, ENDFILE, and REWIND statements are

```
BACKSPACE { u      }
          { din    }
          { (alist)}

ENDFILE   { u      }
          { din    }
          { (alist)}

REWIND    { u      }
          { din    }
          { (alist)}
```

where *u* is an external unit identifier,

din is a dataset identifier whose value specifies the name of an external dataset, and

alist is the following set of specifiers.

```
[UNIT =] u or din
IOSTAT = ios
ERR = s
```

alist must contain a single external unit specifier or dataset specifier and can contain, at most, one of each of the other specifiers. See the UNIT, IOSTAT, and ERR specifiers described for the OPEN and CLOSE statement in table 5-3 and 5-4, respectively.

The external unit or dataset specified by a BACKSPACE or ENDFILE statement must not be connected for direct access. If the external unit or dataset specified by a BACKSPACE, ENDFILE, or REWIND statement is not connected, it becomes connected and the dataset 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 dataset.

The ANSI FORTRAN Standard does not provide for the *din* parameter on BACKSPACE, ENDFILE, or REWIND statements.

BACKSPACE STATEMENT

A BACKSPACE statement causes the dataset related to the specified unit to be positioned at the beginning of the preceding record. If no preceding record exists, the position of the dataset is unchanged. If the preceding record is an endfile record, the dataset is positioned before it.

The ANSI FORTRAN Standard does not provide for backspacing a dataset that is not connected, a dataset that is connected but does not exist, or one that has been written with list-directed format.

ENDFILE STATEMENT

An ENDFILE statement writes an endfile record as the next record of the dataset. The dataset is then positioned after the endfile record.

After the execution of an ENDFILE statement, a BACKSPACE or REWIND statement must reposition the dataset before execution of an input statement. An output statement creates another file on the same dataset. Execution of an ENDFILE statement for a dataset that is connected but does not exist creates the dataset.

The ANSI FORTRAN Standard does not provide for the writing of an endfile on a dataset that is not connected.

REWIND STATEMENT

A REWIND statement causes the specified dataset to be positioned at its initial point. If the dataset is already positioned at its initial point, execution of the statement has no effect on the dataset position.

The ANSI FORTRAN Standard does not provide for the rewinding of an unconnected dataset or a dataset connected for direct access, and it does not provide for the creation of a connected dataset that did not exist.

INQUIRE STATEMENTS

An INQUIRE statement determines the current status of an external dataset's attribute. Inquiry can be made by dataset or by unit.

INQUIRY BY DATASET NAME

The format of the INQUIRE by dataset name statement is

```
INQUIRE (idlist)
```

where *idlist* is a list of inquiry specifiers that must contain exactly one dataset specifier and can contain, at most, one of each of the inquiry specifiers described in table 5-2.

The format of the dataset specifier is

```
FILE=fin
```

where *fin* is a character expression whose value specifies the name of the dataset. The named dataset need not exist or be connected to a unit. *fin* is limited to seven characters, not counting trailing blanks. If *fin* does contain trailing blanks, they are discarded.

INQUIRY BY UNIT

The format of the INQUIRE by unit statement is

```
INQUIRE(iulist)
```

where *iulist* is a list of inquiry specifiers that must contain exactly one external unit specifier and can contain, at most, one of each of the inquiry specifiers described in table 5-2.

The format of the external unit specifier is

```
UNIT=u
```

where *u* is an external unit identifier. (See unit identifiers described earlier in this section.) The unit specified need not exist or be connected to a dataset. If it is connected to a dataset, however, the inquiry includes the connected dataset.

Table 5-2. Inquiry specifiers and their meanings

Specifier	Data Type	Meaning	Input (I) or Return value (RV)
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
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.
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 dataset (non-ANSI).

INQUIRE STATEMENT RESTRICTIONS

A variable or array element that becomes defined or undefined as a result of its use as a specifier must not be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by dataset name statement causes *nmd*, *fn*, *seq*, *dir*, *fmt*, and *unf* (see table 5-2) to be assigned a value only if the value of *fin* is acceptable as a dataset name and if a dataset by that name exists. Otherwise, these specifiers 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 dataset is connected to the unit. Otherwise, these specifiers become undefined.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifiers except *ios* become undefined. *ex* and *od* always become defined unless an error condition occurs.

OPEN STATEMENT

An OPEN statement connects an existing dataset to a unit, creates a dataset that is preconnected, creates a dataset and connects it to a unit, or changes certain specifiers of a connection between a dataset and a unit.

The format of the OPEN statement is

```
OPEN (olist)
```

where *olist* consists of an external unit specifier and, at most, one of each of the other specifiers described in table 5-3.

If a unit is connected to an existing dataset, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in the OPEN statement, the dataset to be connected to the unit is the same as the dataset where the unit is connected.

If the dataset to be connected to the unit does not exist but is the same as the dataset where the unit is preconnected, the specifications in the OPEN statement become a part of the connection.

If the dataset to be connected to the unit is not the same as the dataset where the unit is connected, the effect is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately before the execution of the OPEN statement.

If the dataset to be connected to the unit is the same as the dataset where the unit is connected, only the BLANK= specifier can have a value that is different from the current value. Execution of the OPEN statement causes the new value of the BLANK= specifier to be in effect. The dataset position is unaffected.

If a dataset is connected to a unit, execution of an OPEN statement on that dataset and a different unit is not permitted.

CLOSE STATEMENT

A CLOSE statement terminates the connection of a particular dataset to a unit and rewinds the dataset. The format of a CLOSE statement is

```
CLOSE (cllist)
```

where *cllist* consists of an external unit specifier and, at most, one of each of the other specifiers described in table 5-4.

Execution of a CLOSE statement can occur in any executable program and need not occur in the same program unit as an OPEN statement.

A disconnected dataset or unit can be reconnected within the same executable program either to the same dataset or unit, or to a different dataset or unit, provided the dataset still exists. If the dataset is memory resident, CLOSE deletes the dataset regardless of the STATUS specifier.

The ANSI FORTRAN Standard provides an implicit CLOSE for all datasets upon normal program termination. CFT programs do not perform implicit CLOSEs; datasets do not rewind.

The ANSI FORTRAN Standard does not allow memory resident datasets which are automatically deleted regardless of the STATUS specifier.

Table 5-3. 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
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 method (omitted for sequential 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.

† 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.

††† CFT allows formatted and unformatted records in the same dataset (non-ANSI).

Table 5-4. 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*.

NAMELIST STATEMENT (CFT EXTENSION)

A NAMELIST statement provides a format-free method of specifying input/output lists.

The format of the NAMELIST statement is

```
NAMELIST/group/v[,v]...[[,]/group/v[,v]...]...
```

where *group* is the group name for the following list, and

v is a variable name or an array name. *v* cannot be a dummy argument or a pointee variable.

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. 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.

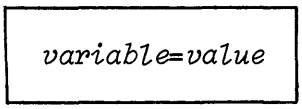
```
READ     (unit,group [,ERR=sn,END=sn])
WRITE    (unit,group [,ERR=sn])
READ     group
PRINT    group
PUNCH    group
```

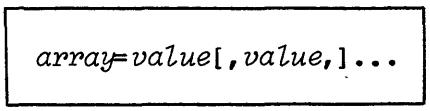
Variable or array names are separated by commas in the NAMELIST statement. These names can be members of more than one NAMELIST group.

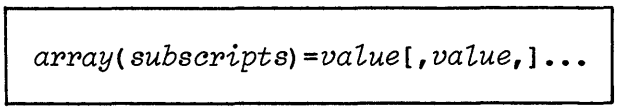
The NAMELIST statement must follow all declaratives affecting the variable or array names and must precede the first use of the group name in any I/O statement.

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.

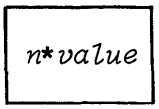






where *subscript* must be an integer constant; multiple array values are assigned in storage order.

Any value can be repeated by



where *n* is the 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.

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 part 2, section 3, 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*]

where *string* is 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.

NAMELIST input processing

The NAMELIST processor scans forward from the current position on the input dataset 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.

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 reasonableness, usefulness, 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 ;.)

NAMELIST OUTPUT

An output NAMELIST group record is written in the following general form.

```
& group name variable name = value,...,
array name = value,...,value,...,&END
```

where *group name*, *variable name*, and *array name* are the 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.

User control subroutines

The following routines provide the user control of the output format.

In the following subroutines, *char* can be any ASCII character specified by 1Lx, or 1Rx. 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

BUFFER IN AND BUFFER OUT STATEMENTS (CFT EXTENSIONS)

Buffered input/output 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 input/output 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.

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.,  DENSITY = 1.,  &END
&OUTPUT  WEIGHT = 9.,  LENGTH = 3.,  WIDTH = 3.,  HEIGHT = 1.,  DENSITY = 1.,  &END
&OUTPUT  WEIGHT = 4.5,  LENGTH = 3.,  WIDTH = 3.,  HEIGHT = 1.,  DENSITY = 0.5,  &END
```

The formats of the BUFFER IN and the BUFFER OUT statements are

```
BUFFER IN ({u  },m)  (bloc,eloc)
          {din}

BUFFER OUT ({u  },m)  (bloc,eloc)
           {din}
```

where *u* is a unit identifier expressed as an integer or as a Hollerith expression of up to seven characters;

din is a dataset name expressed as a character string or a character or integer variable containing a character string of up to seven characters;

m is a mode identifier expressed as an integer expression indicating full record processing if 0 or greater and partial record processing if less than 0;

bloc is the symbolic name of that variable or array element marking the beginning location of the buffered I/O transfer; and

eloc is the 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 dataset *din* and the contiguous memory locations beginning with *bloc* and concluding with *eloc*. If unit *u* or dataset *din* is completing a buffered input/output 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 datasets.

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 specifier, *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.

Dataset and record positioning for buffered input/output operations are as described for non-buffered input/output 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 Library Reference Manual, CRI publication SR-0014.

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 specifier 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
```

THE UNIT FUNCTION (CFT EXTENSION)

After a BUFFER IN or BUFFER OUT statement has been executed, the normal execution sequence continues concurrent with the actual 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 to indicate successful completion of a partial record read operation (BUFFER IN with $m<0$) without encountering the end of the current record,
- -1.0 to indicate successful completion of all other transfers,
- 0.0 to indicate reading of an end-of-file,
- 1.0 to indicate occurrence of a disk parity error during reading, or
- 2.0 to indicate other disk malfunctions 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
```

THE LENGTH FUNCTION (CFT EXTENSION)

If the utility function LENGTH is referenced while a buffered input/output operation is in progress, the execution sequence is delayed until the transfer is complete. LENGTH then provides to the expression in which it is referenced, 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
```

RESTRICTIONS ON INPUT/OUTPUT STATEMENTS

A function must not be referenced in an input/output statement if it causes an input/output statement to be executed.

An input/output statement must not reference a unit or file not having all the properties required for its execution.

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 specifier (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 specifier; 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.

FORMAT SPECIFICATION 6

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 input/output 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.

The ANSI FORTRAN Standard does not limit nesting of format specifications.

FORMAT STATEMENTS

The format of a FORMAT statement is

```
FORMAT fs
```

where *fs* is a format specification.

FORMAT statements must always be labeled.

FORMAT OF A FORMAT SPECIFICATION

The format of a format specification is

([*flist*])

where *flist* is a list in which each list item has one of the forms:

ned
[*r*]*ed*
[*r*](*flist*)

where *ned* is a nonrepeatable edit descriptor,

ed is a repeatable edit descriptor,

r is a nonzero, unsigned integer constant called a repeat specification. If not specified, a value of 1 is assumed.

flist is a format specification with a non-empty list.

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)

                READ(10,FMT='(F10.0)')X
```

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*.

The formats of *repeatable edit descriptors* are

```
[r]Iw[.m]
[r]Fw.d
[r]Ew.d[Ee]
[r]Dw.d
[r]Dw.dEe  (CFT EXTENSION)
[r]Gw.d[Ee]
[r]Ow  (CFT EXTENSION)
[r]Zw  (CFT EXTENSION)
[r]Lw
[r]A[w]
[r]Rw  (CFT EXTENSION)
```

where I, F, E, D, G, O, Z, L, A, and R
indicate the manner of editing,

w, *r*, and *e*
are nonzero, unsigned integer constants, and

d and *m*
are unsigned integer constants.

The *repeat specification*, *r*, optionally precedes any repeatable edit descriptor.

Examples:

```
E20.5E3   3E20.5E3   I3   F8.5   E19.12   D8.1   G13.3
O23   Z6    L7   A8   R6   A   5I9   2F6.0  12E7.2   3D10.0
29G5.0   6O23   2Z10   7L7   3A5   4R4   3A
```

The formats of nonrepeatable edit descriptors are

```
'h1h2...hn'   (apostrophe)
"h1h2...hn"   (quotation mark)   (CFT EXTENSION)
nHh1h2...hn
Tc
TLc
TRc
nX
X   (CFT EXTENSION)
/   (slash)
n/   (slash)   (CFT EXTENSION)
:   (colon)
$   (dollar sign)   (CFT EXTENSION)
kP
BN
BZ
S
SP
SS
```

where apostrophe, quotation mark, H, T, TL, TR, X, slash, colon, P, BN, BZ, S, SP, and SS indicate the manner of editing;

h is any ASCII character listed in Appendix A as capable of internal representation;

c and *n* are positive, nonzero, unsigned integer constants; and

k is an optionally signed integer constant.

Examples:

```
'AN APOSTROPHE EDIT DESCRIPTOR'
"A QUOTATION-MARK EDIT DESCRIPTOR"
T112
55X
/
6/
:
$
3P
TL3
TR6
BZ
SS
```

Table 6-1 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 6-1. 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 6-2 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 6-2. Edit descriptors and data types when SEGLDR and the EQUIV directive are used

Data types	Edit descriptors									
	I	F	E	D	G	L	A	O	Z	R
Integer	*	*	*	-	*	*	*	*	*	*
Logical	*	*	*	-	*	*	*	*	*	*
Real	*	*	*	*	*	*	*	*	*	*

INTERACTION BETWEEN I/O LISTS AND FORMAT SPECIFICATIONS

The beginning of execution of a formatted input/output 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 input/output list, if one exists.

If a statement has an input/output 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 input/output 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 input/output list, except that an item of type complex requires the interpretation of two F, E, D, G, A, or R edit descriptors. An input/output list contains no items corresponding to nonrepeatable edit descriptors.

When format control encounters a repeatable edit descriptor, it determines whether the input/output 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.

When a colon edit descriptor is encountered and no more input/output list items remain to be processed, format control is terminated. Otherwise, the colon edit descriptor is ignored.

Format control also terminates if it encounters the rightmost parenthesis of a complete format specification and if no additional input/output 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))
                      ↑
```

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.

INTERNAL REPRESENTATION

A field is a part of a record that is read or written when format control processes a single repeatable edit descriptor or an H, apostrophe, or quotation mark edit descriptor. *Field width* is the size of the field in characters.

Internal representation of data corresponds to the internal representation of a constant of similar type.

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
```

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')
```

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.

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.

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.

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/,...)
```

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 input/output list items remain to be processed. If unprocessed input/output 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.
```

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
```

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 input/output 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:

Input 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

The scale factor k controls decimal normalization. If $-d<k\leq 0$, 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.

NUMERIC EDITING (BN, BZ, S, SP, SS, I, F, E, D, AND G)

Numeric editing specifies input/output 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.

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= specifier when the unit was opened. NULL (BN) is the default.

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.

Integer editing

The Iw and I$w.m$ edit descriptors indicate that the field to be edited occupies w positions. The specified input/output 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 I$w.m$ 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 I*w*.*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.

F editing

The F*w*.*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 1 2 3 4 5 6 7 8 9 10	F edit descriptor	Internal 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 representation	F edit descriptor	Output field positions 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

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 ... x_{d-1} x_d *exp*

where $x_1x_2 \ldots x_d$ 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*|≤99	E±y_1y_2
E*w*.*d*	100≤|*exp*|≤999	±$y_1y_2y_3$
E*w*.*d*	1000≤|*exp*|≤2466	±$y_1y_2y_3y_4$
E*w*.*d*E*e*	|*exp*|≤(10**e)-1†	E±$y_1y_2y_3 \ldots y_e$

† If *e* is greater than the number of digits necessary to express *exp*, leading zeros are inserted.

An \|*exp*\|≥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 1 2 3 4 5 6 7 8 9 10 11 12	E edit descriptor	Internal 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 representation	E edit descriptor	Output field positions 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

D (double-precision) editing

D editing is identical to E editing.

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 $N \geq 10^{d}$	kP,Ew.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 descriptor	Internal representation
1 2 3 4 5 6 7 8 9 10 11 12		
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 representation	G edit descriptor	Output field positions 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

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.

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.

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.

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 input/output 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 (1 2 3 4 5 6 7 8 9 10 11 12)	L edit descriptor	Internal 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 representation	L edit descriptor	Output field positions (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

A (ALPHANUMERIC) EDITING

The A[w] edit descriptor is used with an input/output list item of type character, logical, integer, real, or complex. w specifies the field width. If w is not specified, the input/output list item must be of type character, in which case the number of characters in the field is the length of the character input/output 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 input/output 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 input/output 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												Item	A edit	Internal
1	2	3	4	5	6	7	8	9	10	11	12	type	descriptor(s)	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

R (RIGHT-JUSTIFIED) EDITING (CFT EXTENSION)

The R*w* edit descriptor is used with an input/output 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.

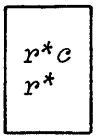
Examples:

Input field positions 1 2 3 4 5 6 7 8 9 10 11 12	Item type	R edit descriptor	Internal 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 representation	Item type	R edit descriptor	Output field positions 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

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 specifier. List-directed records consist of values and value separators. Each value is either a constant, a null value, or one of the following forms.



where r is an 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 be in 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

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. The end of a 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

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 Iw 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 $10**^{-2466} \leq x < 10**^{2466}$. If the magnitude x is within this range, the constant is produced with 0PFw.d; otherwise, 1PEw.dEe 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.

PROGRAM UNIT SPECIFICATION 7

A program unit is a specific collection of FORTRAN statements and comment lines. A program unit is either a main program or a subprogram.

A *main program* is a program unit that does not contain a FUNCTION, SUBROUTINE, ENTRY, RETURN, or BLOCK DATA statement. An optional PROGRAM statement can be the first statement of a main program. An executable program must contain one main program. Program execution begins with the first executable statement of the main program. A main program must not be referenced from a subprogram or from itself.

A *subprogram* is a program unit that can be referenced from a main program or another subprogram. A subprogram begins with either a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

PROGRAM STATEMENT

Although the PROGRAM statement is optional, its use is strongly recommended since several compiler options (for example, F, H) depend on the presence of a PROGRAM statement. When used, it must be the first statement of the main program.

The format of a PROGRAM statement is

```
PROGRAM pgm [(h)]
```

where *pgm* is the symbolic name of the main program where the PROGRAM statement appears, and

h is a sequence of any allowable FORTRAN characters except $.

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

The symbolic name *pgm* is global and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. The name *pgm* must not be the same as any local name in the main program. It can be followed by a parenthesized character string that has no effect on the executable program.

Examples:

```
PROGRAM A1B2C3D4

PROGRAM X (INPUT,OUTPUT)
```

FUNCTION SUBPROGRAMS

FUNCTION statements identify and reference function subprograms. See part 1, section 4 for a description of function subprograms.

FUNCTION REFERENCE

A function reference references an intrinsic function, statement function, or external function. The format of a function reference is

fun([*a*[,*a*]...])

where *fun* is the symbolic name of a function or dummy procedure, and

a is an actual argument.

FUNCTION STATEMENT

The format of a FUNCTION statement is

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

where *type* can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[**len*],

fun is the symbolic name of the function subprogram where the FUNCTION statement appears,

d is a dummy argument representing a variable, array, or dummy procedure name, and

len is the length of the result of a character function. *len* 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, (*).

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 be defined during execution of the procedure. If this variable is a character variable with a length specification of (*), it must not appear as an operand for concatenation except in a character assignment statement.

STATEMENT FUNCTION DEFINITION STATEMENT

The format of a statement function definition statement is

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

where *fun* is the symbolic name of the statement function,

d is a statement function dummy argument, and

e is an expression.

The relationship between *fun* and *e* must conform to the assignment rules in table 3-4, part 1, section 3. The type of the expression *e* can be different from the type of the statement function name *fun*.

Each *d* is a variable name called a *statement function dummy argument*. The names of variables that appear as dummy arguments of a statement function have a scope of that statement only. A given symbolic name can appear only once in a single dummy argument list.

Statement function dummy arguments serve only to indicate the order, number, and type of arguments for a single statement function. The same symbolic names can be used to identify dummy arguments of the same type

in a different statement function definition statement and to identify variables (including dummy arguments of a subprogram) of the same type appearing elsewhere in the program unit. They must not identify any other entity in the program unit except a common block.

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 for which the statement function definition statement appears in preceding lines of the program unit
- An external function reference
- A dummy procedure reference
- An expression enclosed in parentheses

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression portion of a statement function definition statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement can be referenced in the expression of a statement function statement within that subprogram.

Examples:

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

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

CIRCUM(R)=6.2831852*R

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

The length specification of a character statement function or statement function dummy argument of type character must be an integer constant expression.

SUBROUTINE AND CALL STATEMENTS

SUBROUTINE statements identify subroutine subprograms. CALL statements reference subroutine subprograms. See part 1, section 4 for a description of subroutine subprograms.

SUBROUTINE REFERENCE

A subroutine is referenced by a CALL statement. The format of a CALL statement is

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

where *sub* is the symbolic name of a subroutine or dummy procedure, and

a is an actual argument or an alternate return specifier.

Execution of a CALL statement

Execution of a CALL statement results in the following.

- The evaluation of actual arguments that are expressions
- The 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 the alternate return specifier argument. The format of an alternate return specifier is

```
*s
```

where *s* is the statement label of the executable statement located in the same program unit where CALL appears to which control can be returned.

Return of control to the referencing program unit completes the execution of the CALL statement.

Examples:

```
CALL SAM

CALL GEORGE(X,-1)

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

SUBROUTINE STATEMENT

The format of a SUBROUTINE statement is

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

where *sub* is the symbolic name of the subroutine, and

d is a dummy argument representing a variable name, an array name, a dummy procedure name, or an asterisk associated with an alternate return specifier (See the RETURN statement).

Examples:

```
SUBROUTINE SAM

SUBROUTINE GEORGE(A,B)

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

RETURN STATEMENT

A RETURN statement causes control to return to the referencing program unit. The RETURN statement can appear only in a function or subroutine subprogram.

The format of a RETURN statement in a function subprogram follows.

```
RETURN
```

The format of a RETURN statement in a subroutine subprogram is

```
RETURN [e]
```

where *e* is an integer expression indicating an alternate return.

EXECUTION OF A RETURN STATEMENT

Execution of a RETURN statement terminates a reference to a function or subroutine subprogram. Such subprograms can contain more than one RETURN statement. 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. During program execution, a function or subroutine subprogram must not be referenced twice without an intervening RETURN or END statement.

The value of a function must be defined before the execution of its RETURN or END statement. Execution of a RETURN or END statement in a procedure subprogram causes return of control to its referencing program unit.

Return of control to the referencing program unit completes execution of the CALL statement.

If a named common block appears in the main program, the entities in the named common block do not become undefined upon execution of any RETURN or END statement in the executable program.

ALTERNATE RETURN

The alternate return option permits control to be returned to the statement identified by the alternate return specifier in the corresponding CALL statement. The value of *e* references the *e*th asterisk in the dummy argument list of a SUBROUTINE or ENTRY statement.

Asterisks in these dummy argument lists are associated with the alternate return specifiers in the CALL statement. If *e* is less than one or greater than the number of asterisks specified, RETURN *e* is treated as RETURN.

In the following example, execution of statement 10 returns control to statement 5, and execution of statement 11 returns control to statement 6.

Example:

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

ENTRY STATEMENT

The ENTRY statement appears only in a procedure subprogram to permit its being entered at any executable statement not within a DO-loop or IF-block range. A procedure subprogram can contain one or more ENTRY statements following its FUNCTION or SUBROUTINE statement.

The format of an ENTRY statement for a function or subroutine subprogram is

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

where *en* is a function or subroutine name that is an entry in the procedure subprogram, and

d is a dummy argument representing a variable name, array name, dummy procedure name, or an asterisk associated with an alternate return specifier.

An alternate return asterisk can only be used in a subroutine ENTRY statement.

REFERENCING A PROCEDURE SUBPROGRAM ENTRY

Referencing an *en* in a function or subroutine subprogram is the same as referencing the function or subroutine subprogram name. Execution begins with the first executable statement following that ENTRY statement.

The order, number, and types of names appearing as 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. Agreement of type is not required where a dummy argument corresponds to an actual argument specifying a subroutine name or an alternate return specifier since no type is associated with either a subroutine name or an alternate return specifier.

ENTRY ASSOCIATION IN FUNCTION SUBPROGRAMS

The function name *en* specified in an ENTRY statement in a function subprogram is associated with all variables associated with the function name appearing in the FUNCTION statement. When any 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. A function name appearing in a FUNCTION statement can differ in type from function names appearing in ENTRY statements in the same subprogram.

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 for 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.

An asterisk is permitted as a dummy argument only in subroutine subprograms. In a subprogram, a dummy argument specified in an ENTRY statement cannot be referenced in a statement function statement unless it also appears as a dummy argument in the statement function statement or in a preceding FUNCTION, SUBROUTINE, or ENTRY statement.

If a dummy argument name is referenced in an executable statement, it must also be specified in that FUNCTION, SUBROUTINE, or ENTRY statement referenced before execution of the executable statement.

EXTERNAL STATEMENTS

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

The format of an EXTERNAL statement is

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

where *proc* is the 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.

Example:

MAIN is the main program of an executable program that includes the functions STAT, STDEV, and MEAN. Considering just the main program, the syntax in which the symbolic name STAT appears defines it as the name of a function. The names STDEV and MEAN, however, appear in a syntax incapable of defining them as function names. This definition is established by the EXTERNAL STDEV, MEAN statement in the second line.

```
PROGRAM MAIN

EXTERNAL STDEV,MEAN
 .
 .
 .
X = STAT(STDEV,SIGMA)

Y = STAT(MEAN,SIGMA)
 .
 .
 .
END

FUNCTION STAT(OP,VALU)
 .
 .
 .
STAT = OP(VALU)

END
```

```
FUNCTION STDEV(S)
 .
      .
      .
STDEV = RMDS

END

REAL FUNCTION MEAN(S)
           .
           .
           .
     MEAN = AVG

     END
           .
           .
           .
```

BLOCK DATA STATEMENTS

Block data subprograms provide initial values for variables and array elements in named common blocks.

The format of a BLOCK DATA statement is

```
BLOCK DATA [sub]
```

where *sub* is the 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.

During one invocation of CFT, 26 un-named block data subprograms can be encountered. CFT assigns the name LOCKDATA to the first un-named block data subprogram, LOCKDATB to the second, LOCKDATC to the third, etc. Use care with un-named block data routines since the loader does not load multiple routines with the same name, as is caused by separate compilations of two different un-named block data routines. Any number of differently named block data subprograms can be specified in an executable program.

The ANSI FORTRAN Standard allows only one un-named block data subprogram.

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
```

PART 3

THE CFT COMPILER

CFT COMPILER I/O 1

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

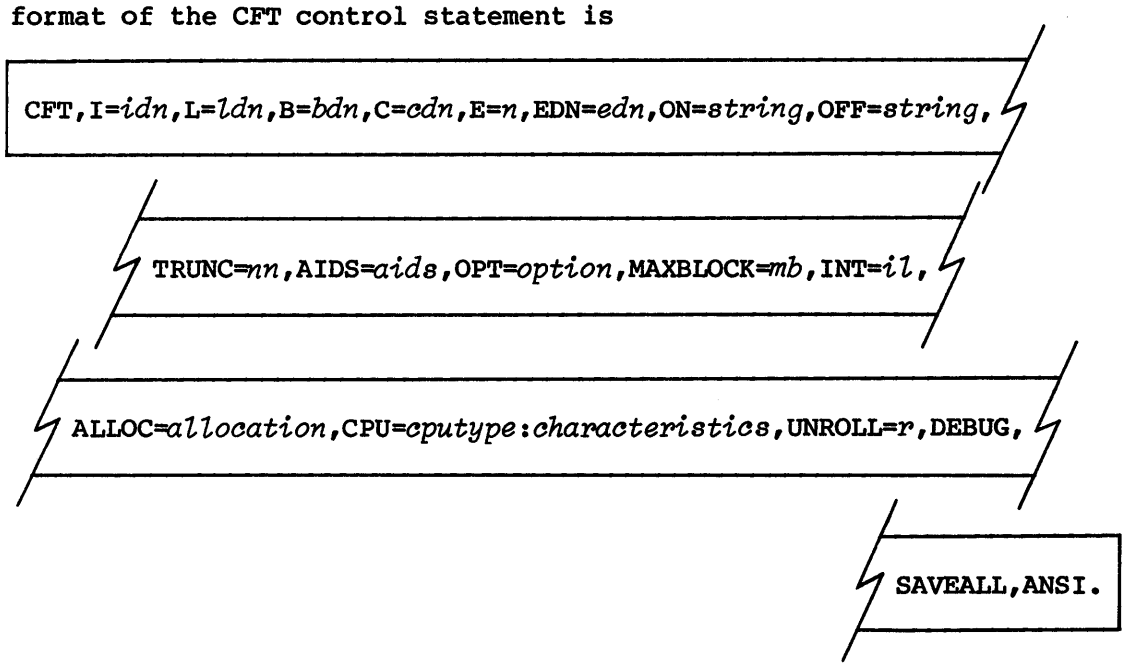
When a CFT control statement is executed, the Cray Operating System (COS) calls upon the system loader to load CFT from the mass storage subsystem. The compiler responds to information in a COS job deck to locate and compile the FORTRAN program. Both binary and symbolic information are output from the compiler.

In this section, CFT is described in terms of its input and output characteristics.

THE CFT CONTROL STATEMENT

The CFT compiler is loaded and executed when a CFT control statement is encountered in the control statement stream.

The format of the CFT control statement is



Options can be in any order. If a keyword and option are omitted from the statement, the compiler uses a default value. 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.

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

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 1-2) specified by the ON= specification are overridden.

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.

E=*n* Level of severity of CFT-produced messages to be listed. The following levels are available. (Also see the ANSI parameter.)

Message level	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)

Message level	Severity type	Description
5	ERROR	Fatal error
n		The highest message levels to be suppressed. 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. (Also see the ANSI parameter.)

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.

ON=*string* Enables compile options (see table 1-2, list of up to 15 characters representing options to be enabled)

OFF=*string*
Disables list or compile options (see table 1-2, list of up to 15 characters representing options to be disabled)

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

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

OPT=*option*
Specifies optimization options. When selecting multiple options, separate values by colons. Option values are:

option	Description
NOZEROINC	Assumes constant increment integers are not incremented by variables with the value 0 (default option)
ZEROINC	Assumes constant increment integers (CII) can be incremented by variables with the value 0. This option inhibits the vectorization of any DO-loop in which there are CIIs of the form CII=CII+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*=CVMG*x*(*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.
FULLIFCON	Allows CFT to optimize conditional replacement statements as described for PARTIALIFCON, except 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 1-line DO-loops with a call to a $SCILIB routine performing the same operation more efficiently (default option). Replacement does not occur when a 1-line DO-loop contains potential dependencies or equivalenced variables.

option	Description
SAFEDOREP (continued)	See part 3, section 2 for examples on the use of SAFEDOREP. DO-loop replacement can be disabled and re-enabled within a program unit by specifying CDIR$ NODOREP and RESUMEDOREP, respectively.
FULLDOREP	Enables replacement of 1-line DO-loops with a call to a $SCILIB routine performing the same operation more efficiently. Potential dependencies and equivalences are ignored. DO-loop replacement can be disabled and re-enabled within a program unit by specifying CDIR$ NODOREP and RESUMEDOREP, respectively.
NODOREP	Disables replacement of 1-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 into the loop preamble (default option)
NOINVMOV	Disables movement of any invariant code from the DO-loop body into the loop preamble
UNSAFEIF	Enables instructions to move over a branch instruction by the instruction scheduler
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 scalar loops to be bottom loaded; intended to obtain correct code where the subscript for a load would be out of range if executed.

option	Description
BTREG	Causes CFT to allocate specific 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 24. If there are fewer than 24 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 24 variables in a program unit, the first 24 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 24 variables at the beginning of a program unit. Variables can be excluded from T registers by specifying their names in a SAVE statement. 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 table 1-1 in this section.) 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 compile loops with specific ambiguous dependencies in vector and scalar versions. A run-time test will determine which version is used. Default is OPT=CVL.
NOCVL	Prevents CFT from compiling loops with ambiguous dependencies in vector and scalar versions. A run-time test will determine which version is used.
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 (see part 3, section 2) 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.

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 or vectorization.

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

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

ALLOC=*allocation*

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

allocation	Description
STATIC	All memory is statically allocated; a stack is not used (default option). (See the SAVEALL parameter, BTREG control statement option, and table 1-1.)

allocation	Description
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.
HEAP	Deferred implementation

CPU=*cputype*:*characteristics*

Specifies mainframe type and optional mainframe characteristics running the generated code; default is the machine running CFT.

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 CRAY X-MP Computer Systems

characteristics	Description
EMA:NOEMA	Target machine does/does not have extended memory addressing
CI:NOCI	Target machine does/does not have compressed index hardware
GS:NOGS	Target machine does/does not have gather/scatter hardware
VPOP:NOVPOP	Target machine does/does not have a vector population count functional unit

Characteristic specification is optional, but it requires a *cputype*. The *cputype* assumes the minimum characteristics for that mainframe. For example, specifying CPU=CRAY-1A does not assume the VPOP characteristic. If the target machine has a vector population count functional unit upgrade, the CFT control statement requires the CPU=CRAY-1A:VPOP specification to use the vector population count hardware.

Unspecified characteristics are assumed to be disabled. For example, if CPU=CRAY-XMP:NOEMA is specified, NOGS and NOCI are assumed to be the minimum set of characteristics for a CRAY X-MP Computer System.

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.

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.

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 table 1-1.)

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.

Table 1-1. Effect of ALLOC, SAVEALL, and BTREG on variable allocation

When options are:			Variable appears in:			
			SAVE, DATA, or COMMON statement†	Array declaratives: CHARACTER COMPLEX DOUBLE EQUIVALENCE NAMELIST†	Other user variables††	Compiler-generated variables†††
ALLOC=	SAVEALL specified	BTREG or NOBTREG				
STATIC (default)	no (default)	NOBTREG (default)	static	static	static	register or static
STATIC (default)	no (default)	BTREG	static	static	register or static	register or static
STATIC (default)	yes	either	static	static	static	register or static
STACK	no (default)	NOBTREG (default)	static	stack	stack	register or stack
STACK	no (default)	BTREG	static	stack	register or stack	register or stack
STACK	yes	either	static	static	static	register or stack

† An entity appearing in (or equivalenced to an entity appearing in) a SAVE, DATA, or COMMON statement is allocated to static memory; otherwise, refer to the array declaratives or other user variable entries in table 1-1.

†† A user variable is allocated to a T register if it appears in table 1-1 and the variable is one of the first 24 variables in the program unit. Local scalar variables not appearing in EQUIVALENCE, DATA, or NAMELIST declaratives are also allocated to T registers.

††† 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. A compiler-generated variable is allocated to a register or memory depending on how the variable is used.

where

register Value allocated to the T register or value or address allocated to the B register (memory is not allocated); values terminated when a END or RETURN is executed.

static Value allocated to Permanent Static Memory maintained as long as the job step exists.

stack Value allocated to Local Stack Memory; values terminated when an END or RETURN is executed.

ERROR MESSAGES DURING PROGRAM EXECUTION

While under control of COS, the executable program calls on system routines to accomplish its mathematical, input/output, and utility operations. These operations are required during compilation of the program. They are loaded from the system or user libraries and linked to the program by the system loader (LDR). When used, the routines respond to programming and/or equipment discrepancies by placing messages in the jobfile and in the $OUT dataset. These discrepancies also cause the job to abort. The COS error messages and descriptions appear in the CRAY-OS Message Manual, publication SR-0039.

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 and the formats specified in part 2. The result is *source code*.

Other information required by CFT to complete its operations is provided by COS 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. (See the subsection on the CFT control statement described earlier in this section.)

Table 1-2. Compiler options

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
J	Causes all DO-loops to be executed at least once	OFF
L	Enables recognition of output listing control directives	ON
N	Enters null symbols in the Symbol Table (defined but not referenced)	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 1-2. Compiler options (continued)

Option	Description	Default
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.	ON
Q	Aborts compilation when 100 fatal error messages counted	ON
R	Rounds results on multiply operation	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
V	Vectorizes inner DO-loops	ON
W	Compiles all floating-point operations as return jumps to user-supplied external routines.†† (See table 1-3.)	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

† 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.

OUTPUT FROM CFT

Relocatable binary output is written on the B-dataset in a format suitable for input by the system loader, LDR, one record per program unit. When requested, LDR loads and links this file plus routines required from the system or user libraries.

LISTABLE OUTPUT

CFT optionally produces a 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 and compiler directives allow the user to control this output and specify the receiving dataset. Listable output is divided into pages. The number of lines per page is controlled by the LPP parameter on the OPTION control statement (see the CRAY-OS Version 1 Reference Manual, publication SR-0011).

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 table 1-2)
- The date and time compilation began
- The CFT revision level and assembly date
- The global page number

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.

BLOCK BEGINS messages

CFT divides program units into smaller units called blocks. These blocks 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.

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.

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

Address field - Values in this field are in octal and are either addresses relative to the beginning of the program, the local stack area, or a named common block; or a B or T register number.

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.

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

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 *.

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, 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

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 part 1, 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.

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.

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 follow the program block and their names begin with a # sign. The program block and pound blocks are created by the compiler and contain code and static data to execute the compiled program unit.

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 number 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.

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.

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.

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

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.)

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.

COMPILER OPTIONS

Compiler options expressed by the user in the CFT control statement (see description of the CFT control statement 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 CFT control statement E (enable compiler directives) option must be specified ON for compiler options to be recognized by CFT. Otherwise, the lines containing compiler options are treated as comment lines.

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 card. 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.

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

LISTABLE OUTPUT CONTROL DIRECTIVES

Following are the listable output control directives.

- EJECT
- LIST
- NOLIST
- CODE
- NOCODE

The CFT control statement options L (listable output control directives) and E (error messages) must be on 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.

The form of the EJECT directive follows.

```
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.

The form of the LIST directive follows.

NOLIST directive

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

The form of the NOLIST directive follows.

CODE directive

The CODE directive produces CFT-generated code listings if previously suppressed by a listing directive or by the CFT control statement OFF=G or OFF=L list option. 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.

The form of the CODE directive follows.

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.

The form of the NOCODE directive follows.

NOTE

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

VECTORIZATION CONTROL DIRECTIVES

The vectorization control directives require the ON=V CFT statement 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 superceded by another vectorization directive.

The form of the VECTOR directive follows.

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 superceded by another vectorization directive.

The form of the NOVECTOR directive is

```
NOVECTOR[=n]
```

where *n* is 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.

The form of the NORECURRENCE directive is

```
NORECURRENCE[=n]
```

where *n* is an integer constant or a previously defined integer parameter in the range 0 to 64.

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*

where S is a scalar variable, and

e is any expression not inhibiting vectorization.

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. DO-loop execution occurs only if the current iteration count for any vector loop is set, by default or a vector directive, less than or equal to *n*. The default value of *n* is 14.

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 15 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.

IVDEP directive

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 whether or not an IVDEP directive is specified. See part 3, section 2 for information on Bidirectional Memory.

The form of the IVDEP directive follows.

```
IVDEP
```

IVDMO directive

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. See part 3, section 2 for information on Bidirectional Memory.

The form of the IVDMO directive follows.

```
IVDMO
```

VFUNCTION directive

The VFUNCTION directive declares that a vector version of an external function exists.

The form of the VFUNCTION directive is

```
VFUNCTION f[,f]...
```

where *f* is 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.

The form of the NEXTSCALAR directive follows.

```
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.

The form of the SHORTLOOP directive follows.

```
SHORTLOOP
```

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.

The form of the integer control directives are

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

where 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 type 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.

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.

FLOW TRACE DIRECTIVES (FLOW/NOFLOW)

Flow trace directives print a summary on dataset $OUT, listing the following information about each subroutine in a program.

- 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 argument passage 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 by using ON=F on the CFT statement or by using a CDIR$ FLOW directive in the source deck. 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 flow trace 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 CFT control statement and by including the FLODUMP control statement.

The FLODUMP control statement follows the EXIT and DUMPJOB control statements.

Example:

```
JOB, ... .
CFT,ON=F.
LDR.
EXIT.
DUMPJOB.
FLODUMP.
   .
   .
   .
```

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

Options

As an additional option, the user can select one or more of the following.

- SETPLIMQ
- ARGPLIMQ
- FLOWLIM

SETPLIMQ - This option enables the flow trace routine to print a line on $OUT 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 runtime 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.

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 control card 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 to 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 control card options. The CDIR$ directives apply to one block at a time and the last directive appearing in a block is the directive used.

DYNAMIC COMMON BLOCK DIRECTIVE (DYNAMIC)

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

The form of the DYNAMIC directive is

```
DYNAMIC b[,b]...
```

where *b* is the name of a previously encountered common block.

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=0 parameter is specified on the CFT control statement and can be controlled by a CDIR$ BOUNDS directive. If ON=0 is not specified, all CDIR$ BOUNDS directives are ignored.

The ON=0 option is global to all program units in the compilation. The BOUNDS directives are local to the progam 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.

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.

The form of the NO SIDE EFFECTS directive is

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

where *f* is the 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.

The form of the ALIGN directive follows.

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 card. Optimization is disabled only for conditional replacement statements appearing after the NOIFCON directive. The NOIFCON directive is ignored if the optimization level is NOIFCON.

The form of the NOIFCON directive follows.

RESUMEIFCON directive

The RESUMEIFCON directive enables optimization of conditional replacement statements at the level specified by the OPT=*option* on the CFT control card. 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 control card 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.

The form of the RESUMEIFCON directive follows.

```
RESUMEIFCON
```

RESUMEDOREP directive

The RESUMEDOREP directive, specified before a DO statement, enables replacement of successive 1-line DO-loops by calling a $SCILIB routine at the level specified by the OPT parameter on the CFT control statement. If OPT=NODOREP, CDIR$ RESUMEDOREP is ignored. Part 3, section 2 has examples on the use of this directive. OPT=SAFEDOREP is the default option.

The form of the RESUMEDOREP directive follows.

```
RESUMEDOREP
```

NODOREP directive

The NODOREP directive disables replacement of 1-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.

The form of the NODOREP directive follows.

```
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 compiler option and can be used to save space for loops with known dependencies.

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. 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.

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.

EXTERNAL ROUTINES

The external routines shown in table 1-3 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
```

CRAY FORTRAN PROGRAMMING 2

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.

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.)

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
- Branches to statements not in the loop
- 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 part 3, section 1 for the conditions under which IF statements of the form

IF(*logical exp*)*var=expression*

are compiled as conditional vector merges.

An IF statement referencing a statement number defined outside the DO-loop or a statement number preceding an IF statement inhibits vectorization.

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).

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 integer (CII) - An integer variable incremented or decremented once during each pass through a DO-loop by an invariant expression. The CII definition must be in a statement which is executed during every DO-loop iteration. The expression defining a CII can reference itself or another CII. The expression must not use operators other than plus or minus or involve expressions containing parentheses. For example,

  CII=CII±(INVARIANT EXPRESSION)

- Vector array reference - An array element where one subscript expression contains one CII reference and where any other subscript expression is an invariant expression. The subscript expression containing the CII must be a linear expression algebraically reducible to the following form

  [±*invariant expression*$_1$*]CII[±*invariant expression*$_2$]

  where the only operators, if any, in invariant expression$_1$ are multiply operators. Using parentheses in the CII 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

  $$\begin{bmatrix} + \textit{ integer variable } + \\ - \textit{ integer constant } * \\ \qquad\qquad\qquad\quad - \end{bmatrix} \left(\textit{integer variable} \begin{bmatrix} + \textit{ integer variable} \\ * \textit{ integer constant} \\ - \end{bmatrix} \right)$$

  where at least one term inside the parentheses is a variable and only one variable is a CII.

  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 CIIs; 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
```

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 2-1 shows the first seven elements of array A in vector and scalar modes.

Table 2-1. 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

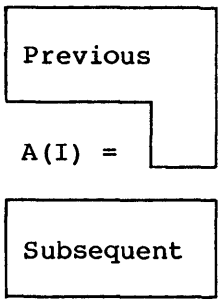
The scalar results are correct by default. CFT detects that the vector results may be different from the scalar results for the DO-loop and inhibits vectorization. A dependency exists if the following two conditions are met.

- 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.



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 2-2 shows dependency information combinations and results.

Table 2-2. 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 CII 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 part 3, 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 part 3, section 1 for a description of IVDEP and other compiler directives.)

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 CII increments, unequal invariant subscripts, and mismatched CIIs 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.

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 CIIs, scalar temporaries, or as recursively defined terms. (CIIs 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 CIIs 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))
```

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.

USE OF OPTIMIZED ROUTINES BY CFT

Several 1-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 Library Reference Manual, CRI publication SR-0014.

1-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 Library Reference Manual, CRI publication SR-0014.

Follow these guidelines for writing 1-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 1-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 1-line DO-loop replacement.

- The DO-loop increment (m_3) must be a positive constant value.

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
                .
                .
                .
```

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.

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.

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 part 3, section 1).

APPENDIX SECTION

CHARACTER SET A

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.

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

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	9-8-7		Substitute
ESC	033	0-9-7		Escape

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
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)
(	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

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
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
R	122	11-9	A,C	Uppercase letter
S	123	0-2	A,C	Uppercase letter

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
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

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
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
{	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

CRAY FORTRAN INTRINSIC FUNCTIONS B

The Cray FORTRAN intrinsic functions described in this appendix have been grouped according to general purpose (that is, trigonometric, exponential, etc.). This grouping is for convenience and is not provided for in the ANSI FORTRAN Standard. These intrinsic functions include all Basic External Functions as described separately in ANSI FORTRAN and Cray FORTRAN extensions.

Entity types in this table are abbreviated: integer (I), real (R), double precision (D), complex (C), logical (L), Boolean (B),† character (CH), 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.

Notations *Full*, *Pseudo*, and *None* indicate the vector status of each subprogram. *Full* indicates the routine uses vector hardware. *Pseudo* indicates the routine uses only scalar hardware but simulates vectorization to gain efficiency for the DO-loop where such a routine appears. *None* indicates only a scalar version exists.

Table B-1 lists generic names and their corresponding specific names. See the Library Reference Manual, CRI publication SR-0014, for additional information on the use of the external routines.

† Boolean functions must be used with caution. If two Boolean results are combined, they are treated as integers.

Function	Definition	ANSI/ CRI extension	Function (y)		Argument(s) (x)			Vector	Code Generated
			Name	Type	No	Type	Range		
GENERAL ARITHMETIC FUNCTIONS									
Truncation	y=[x] Function lost; no rounding	ANSI	AINT DINT	R D	1 1	R D	$\|x\|<2^{46}$ $\|x\|<2^{95}$	Full Full	Inline External
Nearest whole number	y=[x+.5] if $x\geq 0$ y=[x-.5] if $x<0$	ANSI	ANINT DNINT	R D	1 1	R D	$\|x\|<2^{46}$ $\|x\|<2^{95}$	Full Full	Inline External
Nearest integer	y=[x+.5] if $x\geq 0$ y=[x-.5] if $x<0$	ANSI	NINT IDNINT	I I	1 1	R D	$\|x\|<2^{46}$	Full Full	Inline External
Absolute value	y=|x|	ANSI	IABS ABS DABS CABS†	I R D C	1	I R D C	$\|x\|<\infty$ $\|x_r\|,\|x_i\|<\infty$	Full	Inline External
Divide for remainder of x_1/x_2	$y=x_1-x_2[x_1/x_2]$	ANSI	MOD AMOD DMOD	I R D	2 2 2	I R D	$\|x_1\|<2^{63}$, $0<\|x_2\|<2^{63}$, $2^{-63}<\|x_1/x_2\|<2^{63}$ $\|x_1\|<2^{47}$ $0<\|x_2\|<2^{47}$, $2^{-47}<\|x_1/x_2\|<2^{47}$ $\|x_1\|<2^{95}$ $0<\|x_2\|<2^{95}$, $2^{-95}<\|x_1/x_2\|<2^{95}$	Pseudo Full Full	External Inline External
Transfer sign	y= $\|x_1\|$ if $x_2\geq 0$ or y= $-\|x_1\|$ if $x_2<0$	ANSI	ISIGN SIGN DSIGN	I R D	2	I R D	$\|x_1\|$, $\|x_2\|<\infty$	Full	Inline
Positive difference	$y=x_1-x_2$ if $x_1>x_2$ y=0 if $x_1\leq x_2$	ANSI	IDIM DIM DDIM	I R D	2	I R D	$\|x_1\|$, $\|x_2\|<\infty$	Full Full	Inline External
Double-precision product	$y=x_1*x_2$	ANSI	DPROD	D	2	R	$\|x_1\|,\|x_2\|<\infty$	Full	Inline
Imaginary portion of complex value	$y=x_i$	ANSI	AIMAG†	R	1	C	$\|x_r\|$, $\|x_i\|<\infty$	Full	Inline
Conjugate of complex value	$y=x_r-i*x_i$	ANSI	CONJG†	C	1	C	$\|x_r\|$, $\|x_i\|<\infty$	Full	Inline
Obtain random number	y=The first or next in a series of random numbers ($0<y<1$)	CRI	RANF	R	0			Full	External
Obtain random seed	The currently used random number seed	CRI	RANGET	I	1	I	$\|x\|<\infty$	None	External
Establish random number seed	y=x	CRI	RANSET	R	1	B	$\|x\|<\infty$	None	External

† $x=x_r+ix_i$

Function	Definition	ANSI/ CRI extension	Function (y)		Argument(s) (x)			Vector	Code Generated
			Name	Type	No	Type	Range		
EXPONENTIAL FUNCTIONS									
Square root	$y=x^{\frac{1}{2}}$	ANSI	SQRT DSQRT CSQRT†	R D C	1	R D C	$0 \leq x \leq \infty$ $x_r \geq 0,\ x_i < \infty$	Full Full	External
Exponent	$y=e^x$	ANSI	EXP DEXP CEXP†	R D C	1	R D C	$\|x\| < 2^{13} \cdot \ln 2$ $\|x_r\| < 2^{13} \cdot \ln 2$ $\|x_i\| < 2^{24}$	Full Full	External
LOGARITHMIC FUNCTIONS									
Natural logarithm	$y=\ln(x)$	ANSI	ALOG DLOG CLOG†	R D C	1	R D C	$0 < x < \infty$	Full Full	External
Common logarithm	$y=\log(x)$	ANSI	ALOG10 DLOG10	R D	1	R D	$0 < x < \infty$	Full Full	External
TRIGONOMETRIC FUNCTIONS (Angles are in radians)									
Sine	$y=\sin(x)$	ANSI	SIN DSIN CSIN†	R D C	1	R D C	$\|x\| < 2^{24}$ $\|x\| < 2^{48}$ $\left.\begin{matrix}\|x_r\| < 2^{24}, \\ \|x_i\| < 2^{13} \cdot \ln 2\end{matrix}\right\}$	Full	External
Cosine	$y=\cos(x)$		COS DCOS CCOS†	R D C	1	R D C	$\|x\| < 2^{24}$ $\|x\| < 2^{48}$ $\left.\begin{matrix}\|x_r\| < 2^{24}, \\ \|x_i\| < 2^{13} \cdot \ln 2\end{matrix}\right\}$	Full	External
Tangent	$y=\tan(x)$	ANSI	TAN DTAN	R D	1	R D	$\|x\| < 2^{24}$	Full	External
Cotangent	$y=\cot(x)$	CRI	COT DCOT	R D	1 1	R D	$\|x\| < 2^{24}$	Full	External
Arcsine	$y=\arcsin(x)$	ANSI	ASIN DASIN	R D	1 1	R D	$\|x\| \leq 1$	Full	External
Arccosine	$y=\arccos(x)$	ANSI	ACOS DACOS	R D	1 1	R D	$\|x\| \leq 1$	Full	External
Arctangent	$y=\arctan(x)$ $y=\arctan(x_1/x_2)$	ANSI	ATAN DATAN ATAN2 DATAN2	R D R D	1 2 	R D R D	$\|x\| < \infty$ $\|x_1\|, \|x_2\| < \infty$ $\|x_1\|, \|x_2\| \neq 0$	Full	External
Hyperbolic sine	$y=\sinh(x)$	ANSI	SINH DSINH	R D	1	R D	$\|x\| < 2^{13} \cdot \ln 2$	Full	External
Hyperbolic cosine	$y=\cosh(x)$	ANSI	COSH DCOSH	R D	1 1	R D	$\|x\| < 2^{13} \cdot \ln 2$	Full	External
Hyperbolic tangent	$y=\tanh(x)$	ANSI	TANH DTANH	R D	1 1	R D	$\|x\| < 2^{13} \cdot \ln 2$	Full	External

† $x=x_r+i \cdot x_i$

Function	Definition	ANSI/ CRI extension	Function (y)		Argument(s) (x)			Vector	Code Generated
			Name	Type	No.	Type	Range		
MAXIMUM/MINIMUM FUNCTIONS									
Select maximum value	y=The largest of all x	ANSI	MAX0 AMAX1 DMAX1 AMAX0 MAX1	I R D R I	64> n ≥2	I R D I R	$\|x\|<\infty$	Full	Inline
Select minimum value	y=The smallest of all x	ANSI	MIN0 AMIN1 DMIN1 AMIN0 MIN1	I R D R I	64> n ≥2	I R D I R	$\|x\|<\infty$	Full	Inline
CHARACTER FUNCTIONS									
Lexically greater than or equal	$y=a_1 \geq a_2$	ANSI	LGE	L	2	CH	Any legal character string	None	External
Lexically greater than	$y=a_1 > a_2$	ANSI	LGT	L	2	CH	Any legal character string	None	External
Lexically less than or equal	$y=a_1 \leq a_2$	ANSI	LLE	L	2	CH	Any legal character string	None	External
Lexically less than	$y=a_1 < a_2$	ANSI	LLT	L	2	CH	Any legal character string	None	External
Length of character entity		ANSI	LEN	I	1	CH	Any legal character string	None	Inline
Index of a substring		ANSI	INDEX	I	2	CH	Any legal character string	None	External
TYPE CONVERSION									
Conversion to integer	truncation toward zero (fraction lost)	ANSI	INT IFIX IDINT	I I I I I I I	 1 1 1	C I* I R B†† R,B†† D	$\|x_r\|<2^{46}$ $\|x\|<2^{23}$ $\|x\|<2^{63}$ $\|x\|<2^{46}$ $\|x\|<2^{63}$ $\|x\|<2^{46}$	Full	Inline
Conversion to real	$y=x_r^{\dagger}$ y=x y=x accuracy may be lost	ANSI	 REAL FLOAT SNGL	R R R R R R R	 1 1 1	C I* I R I B†† D	$\|x_r\|<2^{46}$ $\|x\|<2^{23}$ $\|x\|<2^{46}$ $\|x\|<\infty$ $\|x\|<2^{46}$ $\|x\|<\infty$	Full	Inline
Conversion to double-precision	$y=x_r^{\dagger}$ accuracy may be extended y=x accuracy may be extended	ANSI	DBLE	D D D D D D	1	C I* I R D B††	$\|x_r\|<\infty$ $\|x\|<2^{23}$ $\|x\|<2^{46}$ $\|x\|<\infty$	Full	Inline
Conversion to complex	y=x $y=x_1 + i \cdot x_2$ or $y=x_1 + i \cdot 0$	ANSI	CMPLX	C C C C C C	1 1 or 2	C I* I B†† R D	$\|x\|<\infty$ $\|x_1\|$, $\|x_2\|<2^{23}$ $\|x_1\|$, $\|x_2\|<2^{46}$ $\|x_1\|$, $\|x_2\|<\infty$	Full	Inline
Character to integer	y=x	ANSI	ICHAR†††	I	1	CH	any legal character	None	External
Integer to character	y=x	ANSI	CHAR§	CH CH	1 1	I B††	0-255	None	Inline
64-bit integer to 24-bit integer	y=x	CRI	INT24	I*	1 1	I B*	$\|x\|<2^{23}$	Full	Inline
24-bit integer to 64-bit integer	y=x	CRI	LINT	I	1	I*	$\|x\|<2^{23}$	Full	Inline

* 24-bit integer

† $x=x_r + i \cdot x_i$

†† Type conversion routines assign the appropriate type to Boolean arguments without shifting or manipulating the bit patterns they represent

††† ICHAR converts a character to an integer based on the character position in the collating sequence

§ CHAR(i) returns the i^{th} character in the collating sequence

Function	Definition	ANSI/ CRI extension	Function (y)		Argument(s) (x)			Vector	Code Generated
			Name	Type	No	Type	Range		
TIME AND DATE									
Real-time clock	Low order 46 bits of clock register expressed as floating point integer Current clock register content	CRI	RTC IRTC	R I	0 0			None	Inline
Time	Current time in ASCII code (*hh:mm:ss*)	CRI	CLOCK†	B	0			None	External
Julian date	Current Julian data in ASCII code (*yyddd*)	CRI	JDATE†	B	0			None	External
Date	Current date in ASCII code (*mm/dd/yy*)	CRI	DATE†	B	0			None	External
BOOLEAN FUNCTIONS									
Logical product	Bit-by-bit logical product (AND) of x_1 and x_2	CRI	AND	B	2	B,I, R,L		Full	Inline
Logical sum	Bit-by-bit logical sum (OR) of x_1 and x_2	CRI	OR	B	2	B,I, R,L		Full	Inline
Logical difference	Bit-by-bit logical difference (XOR) of x_1 and x_2	CRI	XOR	B	2	B,I, R,L		Full	Inline
Equivalence	Bit-by-bit equivalence (XOR) of x_1 and x_2	CRI	EQV	B	2	B,I, R,L		Full	Inline
Not equivalent	Bit-by-bit logical difference (XOR) of x_1 and x_2	CRI	NEQV	B	2	B,I, R,L		Full	Inline
Complement	Bit-by-bit logical complement of x_1	CRI	COMPL	B	1	B,I, R,L		Full	Inline
Mask	x_1=number of one-bits to be left-justified if $0 \leq x_1 < 63$. $(128-x_1)$=number of one-bits to be right-justified if $64 \leq x_1 < 128$	CRI	MASK	B	1	I		Full	Inline
Circular shift	x_1 shifts left x_2 positions; leftmost positions replace vacated positions	CRI	SHIFT	B	2	x_1: B,I, R,L x_2: I	$0 \leq x_2 < 64$	Full††	Inline

† These procedures can be called as subroutines also. An integer or real argument is passed to return the result. However, these procedures cannot be called as both subroutines and functions within a single program unit.

†† The subprogram vectorizes if the second argument is invariant (that is, x_2 is not a vector)

Function	Definition	ANSI/ CRI extension	Function (y)		Argument(s) (x)			Vector	Code Generated
			Name	Type	No	Type	Range		
BOOLEAN FUNCTIONS (cont.)									
Logical shift	x_1 shifts left x_2 positions; leftmost positions lost; rightmost positions set to zero	CRI	SHIFTL	B	2	x_1: B,I, R,L x_2:I	$0 \leq x_2 < 64$	Full††	Inline
	x_1 shifts right x_2 positions; rightmost positions lost; leftmost positions set to zero		SHIFTR	B	2				
Leading zeros	Tallies number of leading zeros in x	CRI	LEADZ	I	1	B,I, R,L		Full	Inline
Population count	Tallies number of ones in x	CRI	POPCNT	I	1	B,I, R,L		Full	Inline†††
Population parity count	0, if x has an even number of ones 1, if x has an odd number of ones	CRI	POPPAR	I	1	B,I, R,L		Full	Inline†††
Merge	Bit-by-bit selective merge $(x_1 \cap x_3) \cup (x_2 \cap \# x_3)$ †	CRI	CSMG	B	3	B,I, R,L		Full§	Inline
VECTORIZATION AIDS									
Vectorization	x_1 returned if $x_3 \geq 0$ x_2 returned if $x_3 < 0$	CRI	CVMGP	B	3	B,I, R,L		Full§	Inline
	x_1 returned if $x_3 < 0$ x_2 returned if $x_3 \geq 0$		CVMGM						
	x_1 returned if $x_3 = 0$ x_2 returned if $x_3 \neq 0$		CVMGZ						
	x_1 returned if $x_3 \neq 0$ x_2 returned if $x_3 = 0$		CVMGN						
	x_1 returned if x_3 is true x_2 returned if x_3 is false		CVMGT			x_1, x_2: B,I, R,L x_3:L			
MISCELLANEOUS FUNCTIONS									
Location	Returns memory address of specified variable or array	CRI	LOC	I	1	B,I,R, L,C,D		None	Inline
Tallies number of arguments used to call subprogram	y=x	CRI	NUMARG	I	0			None	Inline

† The logical symbol ∩ represents AND, the logical symbol ∪ represents OR, and the logical symbol ∩# represents AND NOT.
†† The subprogram vectorizes if the second argument is invariant (that is, x_2 is not a vector).
††† The CRAY-1 A and CRAY-1 B Computer Systems will be external without population parity hardware.
§ The function cannot be passed as an argument.

Table B-1. Generic and specific intrinsic function names

Generic name	Specific names
INT	INT, IFIX, IDINT
REAL	REAL, FLOAT, SNGL
DBLE	†
CMPLX	†
†	ICHAR
†	CHAR
AINT	AINT, DINT
ANINT	ANINT, DNINT
NINT	NINT, IDNINT
ABS	IABS, ABS, DABS, CABS
MOD	MOD, AMOD, DMOD
SIGN	ISIGN, SIGN DSIGN
DIM	IDIM, DIM DDIM
†	DPROD
MAX	MAX0, AMAX1, DMAX1
†	AMAX0, MAX1
MIN	MIN0, AMIN1, DMIN1
†	AMIN0, MIN1
†	LEN
†	INDEX
†	AIMAG
†	CONJG
SQRT	SQRT, DSQRT, CSQRT
EXP	EXP, DEXP, CEXP
LOG	ALOG, DLOG, CLOG
LOG10	ALOG10, DLOG10
SIN	SIN, DSIN, CSIN
COS	COS, DCOS, CCOS
COT††	COT, DCOT
TAN	TAN, DTAN
ASIN	ASIN, DASIN
ACOS	ACOS, DACOS
ATAN	ATAN, DATAN
ATAN2	ATAN2, DATAN2
SINH	SINH, DSINH
COSH	COSH, DCOSH
TANH	TANH, DTANH
†	LGE, LGT, LLE, LLT

† No generic name or no specific name
†† A function not specified by the ANSI standard

CRAY FORTRAN UTILITY PROCEDURES C

The Cray FORTRAN Compiler (CFT) includes a set of utility procedures which, like intrinsic functions, is 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.

Table C-1 describes currently available utility procedures. Entities in this table are abbreviated, 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.

Notations *Full*, *Pseudo*, and *None* indicate the vector status of each subprogram. *Full* indicates the routine uses vector hardware. *Pseudo* indicates the routine uses only scalar hardware but simulates vectorization to gain efficiency for the DO-loop where such a routine appears. *None* indicates only a scalar version exists.

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 utility procedures

Category	Subroutine or Function Name	Func. Type	Description	Argument Type(s) x_1	x_2	x_3	Vector	Code Generated
TIME	TIMEF	R	Returns current clock register content in milliseconds				None	External
	†SECOND	R	x_1 = Cumulative CPU time for job in seconds.	R			None	External
SYSTEM	SYSTEM	I	Selects a COS function. See Exchange Processor, CRI publication SM-0040.	I			None	External
	EXIT		Terminates job step. Job continues at next control statement.				None	External
	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	I		None	External
	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 x_1 is present, it is written to the logfile before the traceback. Argument x_1 is subject to the same restrictions as the argument in REMARK2.	B,I, D,C,L			None	External
	ERREXIT		Terminates job step after program detects error. Gives error exit to COS. Job continues at next EXIT control statement. Prints no traceback.				None	External
	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			None	External
	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 (zero). x_1 = 8 words maximum.	B,I, D,C,L			None	External
	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			None	External
	REMARKF		Enters a message in the logfile. A format and up to 12 variables can be passed, each occupying only one word. The first argument is the format label.	I,C	I,R, L,B		None	External
	SENSEFI		Determines current interrupt mode. Mode=1, interrupts permitted; Mode=0, interrupts prohibited.	I			None	External
	CLEARFI		Temporarily prohibits floating-point interrupts				None	External
	SETFI		Temporarily permits floating-point interrupts				None	External
	CLEARFIS		Permanently prohibits floating-point interrupts				None	External
	SETFIS		Permanently permits floating-point interrupts				None	External
INPUT/OUTPUT	EODW		Writes *eod* and, as required, *eof* and *eor* record(s) on unit x_1	I††			None	External
	IEOF	I	Returns -1 if *eod* processed at unit x_1; +1 if *eof* read at unit x_1; 0, otherwise.	I††			None	External

† These procedures can be used either as functions or as subroutines, but not both within the same program unit.
†† 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 utility procedures (continued)

Category	Subroutine or Function Name	Func. Type	Description	Argument Type(s) x_1	x_2	x_3	Vector	Code Generated
INPUT/OUTPUT (continued)	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††			None	External
	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††			None	External
	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††			None	External
	GETPOS	B	Returns standing address of current record in dataset associated with unit x_1	I††			None	External
	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.	I††	I		None	External
DEBUGGING AIDS	ENDRPV		Resumes error exit processing suspended by SETRPV				None	External
	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.	H	I,R,L	I,B	None	External
	SYMDEBUG		Dumps the contents of specified program variables. The character string argument is identical to the parameters for the DEBUG control statement. See the CRAY-OS Version 1 Reference Manual, publication SR-0011.	H			None	External
	DUMPJOB		Creates an unblocked dataset containing the user job area image (including register states) suitable as input to the DUMP programs	I††			None	External

† These procedures can be used either as functions or as subroutines, but not both within the same program unit.
†† x_1 can be a unit number or a Hollerith unit name. If no argument is specified for DUMPJOB, $DUMP is used.

CFT MESSAGES D

The Cray FORTRAN compiler (CFT) produces six types of compile-time messages. From least severe to most severe, they are

COMMENT - Message reports a programming inefficiency.

NOTE - This usage may cause problems with other compilers.

CAUTION - Message reports a possible user error.

WARNING - Message reports a probable user 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.

- After any other statement - 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.

COMPILE-TIME MESSAGES

"*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*" 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.

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.

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.

SCHARACTER 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.

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.

SCOMMA 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.

SCOMMA 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 *nnnnnnn*
A dependency exists involving an array reference in sequence number *nnnnnnn* 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 seven or eight characters. The ANSI FORTRAN Standard provides for a maximum of six 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 cards encountered or the first card of a program unit is a continuation card.

$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.

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 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 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 $\leq$ C1 $\leq$ C2 $\leq$ 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.

§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 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 CII 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.

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 card 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 CFT control card, MAXBLOCK is set to 1.

CF019 - WARNING: Z WILL BE SET TO ON
When compiling with DEBUG on the CFT control card, Z is forced on.

CF020 - WARNING: I WILL BE SET TO ON
When compiling with DEBUG on the CFT control card, 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.

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 = 1,100
      B(I) = A(3)
      A(I) = 3.0
   55 CONTINUE

      DO 60 I = 1,100
      B(I) = A(I+N)
      A(I) = 3.0
   60 CONTINUE

      DO 70 I = 50,1,-1
      B(I) = A(I,I)
      A(I,3) = 2.0
   70 CONTINUE

      DO 80 I = 1,N,2
      B(I) =  A(3*I-1)
      A(2*I) = 2.0
   80 CONTINUE
```

NULL DEPENDENCY WITH CII 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 CII

```
      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 CII WAS FOUND IN ARRAY REFERENCE

```
      DO 55 I = 1,N
      A(I) = 1.0
      B(J) = B(K)
   55 CONTINUE
```

OUTMODED FEATURES E

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
Random I/O operations	Direct access RECL parameter in I/O list of OPEN and REC parameter in READ or WRITE
DATA statement with declaratives	DATA statement after other declaratives
EOF, IEOF, and IOSTAT functions	End-of-file specifier (END=) or status specifier (IOSTAT=)

HOLLERITH CONSTANTS

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. All Hollerith constants with R suffixes are limited to a maximum of eight characters. An all-zero computer 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.

*n*H...

*n*L...

*n*R...

'...' $\left\{\begin{matrix}H\\L\\R\end{matrix}\right\}$

"..." $\left\{\begin{matrix}H\\L\\R\end{matrix}\right\}$

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.

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 contains eight characters in its first storage unit (computer word) and can contain the value zero or an additional eight characters in its second. 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.

HOLLERITH RELATIONAL EXPRESSIONS

The form of a Hollerith relational expression is

$$e_1 \; relop \; e_2$$

where e_1 and e_2 are Hollerith expressions, and

relop is a 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.

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.

TWO-BRANCH ARITHMETIC IF STATEMENTS

The form of a 2-branch arithmetic IF statement is

$$\boxed{\text{IF}(e)s_1,s_2}$$

where e is an integer, real, or double-precision expression, and

s_1,s_2 are 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
```

INDIRECT LOGICAL IF STATEMENTS

The form of an indirect logical IF statement is

$$\boxed{\text{IF}(e)s_1,s_2}$$

where *e* is a logical expression, and

s_1,s_2 are 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
```

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.

ENCODE AND DECODE STATEMENTS

The forms of the ENCODE and DECODE statements are

```
ENCODE (n,f,dent) [elist]
DECODE (n,f,sent) [dlist]
```

where *n* is the number of characters to be processed, specified by a nonzero integer expression not to exceed 152;

f is a FORMAT identifier, except for an asterisk;

dent is the symbolic name of a destination variable, array element, or array where the *n* characters of *elist* are packed (eight per word) by ENCODE;

sent is the symbolic name of the source variable, array element, or array where characters are unpacked and stored into *dlist* by DECODE; and

elist and *dlist*
are 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.

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'
```

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'
```

EDIT DESCRIPTORS

The formats of obsolete edit descriptors are

$*h_1h_2...h_n*$ (asterisk)

[-*b*]X

where the asterisk and the X indicate the manner of editing,

h_i is any ASCII character listed in Appendix A as capable of internal representation, and

b is any nonzero, unsigned integer constant.

Examples:

```
*AN ASTERISK EDIT DESCRIPTOR*

-55X  (moves current position 55 spaces to left)
```

DOUBLE DECLARATION STATEMENTS

The form of the double declaration type statement is

```
DOUBLE v[,v]...
```

where DOUBLE specifies the desired data type, and

v is a constant, variable, array, function, or dummy procedure name, or is an array declarator.

The form of the double declaration FUNCTION statement is

```
DOUBLE FUNCTION fun[([d[,d]...])
```

where *fun* is the symbolic name of the function subprogram in which the FUNCTION statement appears, and

d is a dummy argument representing a variable, array, or external procedure name.

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.

PUNCH STATEMENT

The PUNCH statement is a data transfer output statement. The format is

```
PUNCH f [,iolist]
```

where *f* is a format identifier, and

iolist is an input/output list specifying the data to be transferred.

TYPE STATEMENT DATA LENGTH

The forms of the type statements with data length included are

```
type [*n] v[,v]...
```

```
IMPLICIT type[*n](a1[-an][,a1[-an]]...)[,type[*n](a1[-an]
    [,a1[-an]]...)]...
```

where *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 is a constant, variable, array, function, or dummy procedure name, or is an array declarator; and

a is 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-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.					

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 non-sequential input/output operations on a dataset with records of various lengths.

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 CRAY-OS Version 1 Reference Manual, publication SR-0011.

DATASET CONNECTION

A dataset is connected for random access through the ASSIGN control statement described in the CRAY-OS Version 1 Reference Manual, publication SR-0011. CFT supports two methods of reading or writing on random access datasets: GETPOS/SETPOS and READMS/WRITMS.

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 Library Reference Manual, CRI publication SR-0014.

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 ($1 \leq NRECS \leq 100$)
- RLENGTH(i) to the number of words in the ith record written ($0 \leq RLENGTH(i) \leq 10$)
- RECORD(j) to the jth word to be written in each non-empty record ($1 \leq j \leq RLENGTH(i)$)

```
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
```

READMS/WRITMS routines

The READMS/WRITMS routines provide a means to create, change, and extend random access datasets. The READMS/WRITMS routines automatically maintain an index and allow records to be rewritten with a different length. It is the user's responsibility to provide the READMS/WRITMS

package with an array to use as an index workspace. Any number of records can be on the dataset but the length of the index array must be sufficient to accommodate all of the records.

Two types of indexing are available: numbered and named. With a numbered index, each record is identified by an integer ordinal. The length of the index array must be at least as large as the maximum number of records. For named indexes, each record is identified by a 64-bit value, typically an ASCII character string. The length of the index array must be at least twice the maximum number of records.

Use of the READMS and WRITMS routines must begin with a call to OPENMS to initialize the index and end with a call to CLOSMS to write the index on the dataset.

See the Library Reference Manual, CRI publication SR-0014 for more information on using the random access dataset management routines CLOSMS, FINDMS, OPENMS, READMS, STINDX, and WRITMS.

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.

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.

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.

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.

CREATING NON-FORTRAN PROCEDURES F

Function and subroutine subprograms written in the Cray Assembly Language (CAL) can be used with FORTRAN programs if proper linkage conventions are followed. Conventional practices linking CAL routines with FORTRAN programs are defined in the following areas.

- Argument transmission
- B and T register use
- Entry block design
- Argument retrieval
- Local temporary variable reference
- Error traceback
- A, S, V, VL, and VM register use
- Function values

Specific macros are available to aid CAL programmers in writing routines following CFT conventions. These macros maintain compatibility with different CFT versions. CFT version 1.11 introduces a new calling sequence. Both new and old calling sequences are available with CFT version 1.11. With CFT version 1.12, only the new calling sequence will be available. The CFT linkage macros are described in the Macros and Opdefs Reference Manual, CRI publication SR-0012.

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.
LDR.
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. For example:

```
CALL SYMDEBUG('O=OUT,PAGES=17.')
```

The string must terminate with a period.

See the CRAY-OS Version 1 Reference Manual, publication SR-0011 for details of the optional DEBUG parameters.

UNBLOCKED DATASETS H

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 CRAY-OS 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.

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 CRAY-OS Version 1 Reference Manual, Appendix F, 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.

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. The format is

```
CALL SETRPV(recname,xpsave,class)
```

where *recname* is the name of the external subroutine to be called if a reprievable error occurs. *recname* must be declared in an EXTERNAL statement.

xpsave is an 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 CRAY-OS Version 1 Reference Manual for a description of *xpsave*.) *xpsave* must be in a common block in the routine that calls SETRPV and in *recname*.

class is a mask that defines recoverable errors. Valid mask values are listed in Appendix F of the CRAY-OS Version 1 Reference Manual, publication SR-0011.

Within *recname*, any FORTRAN statements except RETURN or END can be executed.

REPRIEVE TERMINATION

The normal FORTRAN method of terminating reprieve processing is to call the subroutine ENDRPV from the external subroutine set up by SETRPV. The format follows.

```
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.

FTREF UTILITY J

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 CRAY-OS Version 1 Reference Manual, publication SR-0011, for details of the FTREF control statement.

INDEX

INDEX

A edit descriptor, (2)6-21
A (alphanumeric) editing, (2)6-21
Access
    direct
        dataset position, (2)5-5
        operations, (2)5-5
    random
        dataset creation for, E-13
        modifying a record under, E-15
        positioning while connected for, E-13
    sequential
        dataset position, (2)5-5
        operations, (2)5-4
Active DO-loop, (2)4-11
Actual
    arguments
        association, (1)4-10
        character substring, (1)4-11
        description, (1)4-9, (1)4-10
        external functions, (1)4-5
        subroutine reference, (1)4-2
    array
        declarator, (1)2-11, (1)2-15
        description, (1)2-15
Address field, (3)1-16
Addressing, extended memory (EMA), (1)4-15
Adjustable
    array
        declarator, (1)2-11
        description, (1)2-16
    dimensions, (1)2-16
Aids, vectorization, B-6
ALIGN directive
    description, (3)1-37
    format, (3)1-37
Allocation, memory, (3)1-7
Alphanumeric
    character, (1)1-3
    editing, (2)6-21
Alternate return, (2)7-7
American National Standards Institute, (ANSI), (1)1-1
ANSI
    character set, A-1
    FORTRAN
        character set, A-1
        conformance to standard, (1)1-1
    standards, deviations from, (1)1-2
    X3.9-1978, (1)1-1
Apostrophe and quotation mark editing, (2)6-7
ARGPLIMQ flow trace routine option, (3)1-32
Argument
    actual
        description, (1)4-9, (1)4-10
        external functions, (1)4-5
        in a subroutine reference, (1)4-2
        is a character substring, (1)4-11
    association of dummy and actual arguments, (1)4-10
    description, (1)4-9
    dummy
        array, (1)4-12
        description, (1)4-9
        procedures, (1)4-12
        statement function, (2)7-3
        type character, (1)4-11
        undefined, (1)4-11
        variable, (1)4-11
    retrieval, F-1
    transmission, F-1
Arithmetic
    assignment statement
        execution, (2)3-1
        format, (2)3-1
    constant
        description, (1)2-3
        expression, (1)3-1
    expression
        data type, (1)3-6
        description, (1)3-1
        evaluation, (1)3-10
        forms, (1)3-5
    functions, general, B-2
    IF statement
        execution, (2)4-4
        format, (2)4-4
        two-branch, format, E-6
    operands, (1)3-3
    operators
        description, (1)3-2
        interpretation in expressions, (1)3-2
        precedence, (1)3-3
    relational expression, (1)3-12
Arrangement and reference, array element (1)2-14
Array
    actual, (1)2-15
    adjustable, (1)2-16
    bounds checking directive (BOUNDS), (3)1-34
    declarator
        actual, (1)2-11, (1)2-15
        adjustable, (1)2-11

Array (continued)
assumed-size, (1)2-11
description, (1)2-10
dummy, (1)2-11, (1)2-15
format, (1)2-10
types, (1)2-11
description, (1)2-10
dimensions, (1)2-10
dummy, (1)2-15
dummy argument, (1)4-12
element
arrangement and reference, (1)2-14
data type, (1)2-2
defined, (1)2-21
description, (1)2-10
initially defined, (1)2-21
invariant, description, (3)2-2.1
invariant, used in a reduction array operation, (3)2-4
undefined, (1)2-21
element name
description, (1)2-10, (1)2-12
format, (1)2-12
in an EQUIVALENCE statement, (2)2-12
element order, (1)2-13
equivalenced, (3)2-9
name
description, (1)2-10
in an EQUIVALENCE statement, (2)2-12
use of, (1)2-17
noncharacter, for format specification, E-15
reference, vector, (3)2-3
size, (1)2-12
storage sequence, (1)2-12.1, (1)2-13
symbolic name, (1)2-26
vectorization with, (3)2-10
ASCII
character set, A-1
internal code, (1)1-3
ASSIGN statement
execution, (2)3-3
format, (2)3-3
Assigned GO TO statement
execution, (2)4-3
format, (2)4-3
Assignment statements
arithmetic
execution, (2)3-1
format, (2)3-1
ASSIGN, (2)3-3
character
execution, (2)3-2
format, (2)3-2
description, (2)3-1
formatted data, E-7
logical
execution, (2)3-2
format, (2)3-2
type conversion, (1)3-8
Associated
entities
partially, (1)2-19
totally, (1)2-19
storage sequences, (1)2-19
Association
actual arguments, (1)4-10
common, (2)2-15
description, (1)2-19
dummy arguments, (1)4-10
entities
description, (1)2-19
restrictions, (1)4-13
entry, in function subprograms, (2)7-9
equivalence, (2)2-12
statements, (2)2-11
Assumed-size array declarator, (1)2-11
A, S, V, VL, and VM registers, F-1
Auxiliary character set, (1)1-4, A-1

BACKSPACE statement
description, (2)5-17
format, (2)5-16
B and T register use, F-1
Basic real constant, (1)2-5
Bidirectional memory, (3)2-17
BL directive, (3)1-33
Blank common block, (2)2-15
BLOCK BEGINS messages, (3)1-15
Block
common
blank, (2)2-15
description, (1)4-14
directive, dynamic (DYNAMIC), (3)1-34
extended memory, (1)4-15
name, (1)2-25
named, (1)4-1, (2)2-15
size, (2)2-14
storage sequence, (2)2-14
symbolic name, (1)2-25
task, (1)4-15
data
subprogram, (1)2-26, (1)4-1, (2)7-11
symbolic name, subprogram, (1)2-26
design, entry, F-1
field, (3)1-17
names and lengths in octal, table of, (3)1-18
statement, conditional
description, (2)4-5
execution, (2)4-8
BLOCK DATA statement, (1)4-1, (2)7-11
BLOCK directive, (3)1-36
Block IF statement
execution, (2)4-6
format, (2)4-6
BN and BZ
edit descriptors, (2)6-13
editing, (2)6-13
Boolean (octal or hexadecimal) constant, (1)2-8
Boolean (masking) expression, (1)3-17
Boolean
constant, (1)2-8
data, (1)2-8
expression, (1)3-6
functions, B-5
type conversion, (1)3-10

Bounds, array
checking directive (BOUNDS), (3)1-34
BOUNDS
directive, (3)1-34
options, (3)1-34
BUFFER IN statement (CFT extension)
description, (2)5-29
format, (2)5-31
BUFFER OUT statement (CFT extension)
description, (2)5-29
format, (2)5-31

CAL instructions, EFI and DFI, (3)2-15
CALL ENDRPV format, I-2
CALL SETRPV format, I-1
CALL statement
execution, (2)7-5
format, (2)7-5
Categories, entity, (3)2-2.1
CFT
compiler, (1)1-1, (3)1-1, (3)2-1, D-1
control statement, format, (3)1-1
extensions
BUFFER IN statement, (2)5-29
BUFFER OUT statement, (2)5-29
dollar sign editing, (2)6-10.1
hexadecimal (Z) editing, (2)6-20
IMPLICIT NONE statement, (2)2-10
LENGTH function, (2)5-34
NAMELIST statement, (2)5-23
O (octal) editing, (2)6-19
octal (O) editing, (2)6-19
POINTER statement, (2)2-3
R (right-justified) editing, (2)6-23
UNIT function, (2)5-33
Z (hexadecimal) editing, (2)6-20
input, (3)1-11
language
character set, (1)1-3
definition, (1)1-1
elements, (1)1-3
messages, D-1
output, (3)1-14
use of optimized routines by, (3)2-12
Z option, G-1
Chaining operation, (3)2-18
Character
alphanumeric, (1)1-3
assignment statement
execution, (2)3-2
format, (2)3-2
constant
description, (1)2-8.1
length, (1)2-9
expression evaluation, (1)3-11
expressions, (1)3-11
functions, B-4
primary, (1)3-11
print control, (2)5-15
relational expression, (1)3-13
set
ANSI FORTRAN, A-1
ASCII, A-1
Character (continued)
auxiliary, (1)1-4, A-1
Cray FORTRAN, A-1
description, (1)1-3
FORTRAN, (1)1-3, A-1
special, (1)1-4
substring
actual argument, (1)4-11
description, (1)2-18
type, dummy argument, (1)4-11
CHARACTER type statement
description, (2)2-8
format, (2)2-8
Characteristics, mainframe (3)1-8
Checking, array bounds, directive (BOUNDS), (3)1-34
CII (constant increment integer), (3)2-2.1
cilist (control information list), format, (2)5-9
Classes of symbolic names, (1)2-25
CLEARFI library routine, (3)2-15
CLOSE
specifiers and their meanings, (2)5-23
statement
execution, (2)5-21
format, (2)5-21
CMR (complete memory references) instruction, (3)2-18, (3)2-19
Code
internal, ASCII, (1)1-3
source, (3)1-11
CODE directive
description, (3)1-23
format, (3)1-23
Collating sequence, (1)1-3
Colon
description, (2)6-10
editing, (2)6-10
Comment line, (1)1-9
Common
association, (2)2-15
block
blank, (2)2-15
description, (1)2-25, (1)4-14
directive, dynamic (DYNAMIC), (3)1-34
extended memory, (1)4-15
name, (1)2-25
named, (1)4-1, (2)2-15
size, (2)2-14
storage sequence, (2)2-14
symbolic name, (1)2-25
task, (1)4-15
COMMON statement
blank common, (2)2-15
common association, (2)2-15
description, (2)2-13
format, (2)2-13
restrictions, (2)2-15
size, (2)2-14
storage sequence, (2)2-14

Compiler
definition, (1)1-1, (3)1-1, (3)2-1, D-1
directive line, (1)1-10, (3)1-21
directives, (3)1-20, (3)1-21
options, (3)1-12, (3)1-20
Compile-time messages, D-2
Complete memory references (CMR)
instruction, (3)2-18, (3)2-19
Complex
constant
description, (1)2-7
range, nonzero, (1)2-7
data, (1)2-6, (2)6-19
editing, (2)6-19
expression, (1)3-6
type conversion, (1)3-10
COMPLEX type statement, (2)2-7
Compressed index references, (3)2-16
Computed GO TO statement
execution, (2)4-2
format, (2)4-2
Computer system
CRAY-1, (1)1-1
CRAY X-MP, (1)1-1, (3)2-17
Conditional
block statement
description, (4)4-5
execution, (2)4-8
replacement statement, (3)2-13
statements, (3)2-13
vector loops, (3)2-10
Conditions
error and end-of-file, (2)5-15
inhibiting vectorization, (3)2-2
Conformance with the ANSI standard, (1)1-1
Connection, dataset, E-13
Constant
arithmetic, (1)2-3
basic real, (1)2-5
Boolean (octal or hexadecimal), (1)2-8
character
description, (1)2-8.1
length, (1)2-9
complex, (1)2-7
description, (1)1-5, (1)2-2
double-precision, (1)2-6
expression, arithmetic, (1)3-1
followed by a
double-precision exponent, (1)2-6
real exponent, (1)2-5
Hollerith, E-2
increment integer (CII), (3)2-2.1
integer, (1)2-3, (1)2-4
logical
description, (1)2-7
representation, (1)2-7
nonzero
complex, (1)2-7
double-precision, (1)2-6
real, (1)2-5
optionally signed, (1)2-3
range
nonzero complex, (1)2-7
nonzero double-precision, (1)2-6
nonzero real, (1)2-5
Constant (continued)
real, (1)2-4
signed, (1)2-3
symbolic name, (1)2-27
unsigned, (1)2-3
Continuation line, (1)1-9
CONTINUE statement
execution, (2)4-14
format, (2)4-14
Control
character, print, (2)5-15
directives
integer (INT24, INT64), (3)1-29
listable output, (3)1-22
vectorization, (3)1-24
format, positioning, (2)6-7
information list (*cilist*), format
(2)5-9
processing, loop, (2)4-13
statement
CFT, format, (3)1-1
program, (2)4-1
subroutines, user, (2)5-26, (2)5-28
Conventions, (1)1-2
Conversion
integer length, B-4
type
Boolean, (1)3-10
complex, (1)3-10
description, (1)3-9
double-precision, (1)3-9
functions, B-4
in assignment statements, (1)3-8
integer, (1)3-9
real, (1)3-9
uppercase/lowercase, (1)1-5
COS (Cray Operating System), (1)1-1
Count, iteration, (2)4-12
CRAY-1 Computer System, (1)1-1
Cray FORTRAN (CFT)
character set, A-1
compiler, (1)1-1, (3)1-1, (3)2-1
intrinsic functions, B-1
language, (1)1-1
programming, (3)2-1
utility procedures, C-1
Cray Operating System (COS), (1)1-1
CRAY X-MP Computer System, (1)1-1, (3)2-17
Creating
a dataset for random access, E-13
a file, (2)5-3
non-FORTRAN procedures, F-1
Cross-reference information, (3)1-19
CVL directive, (3)1-39

D editing, (2)6-17
Data
assignment, formatted, E-7
block, subprogram, (1)2-26, (1)4-1,
(2)7-11
Boolean, (1)2-8
complex, (1)2-6, (2)6-19
description, (1)2-1

Data (continued)
double-precision, (1)2-5
editing, (2)6-19
integer, (1)2-3
item, integer, (1)2-3
length
type statement, format, E-11
with data types, E-12
logical, (1)2-7
real, (1)2-4
specification, (2)2-1
association, (2)2-1
declaration, (2)2-1
initialization, (2)2-1
statements, (2)2-1
type, (2)2-1, (2)2-7
transfer, (2)5-11
dataset position before, (2)5-5
description, (2)5-12, (2)5-13
direction of, (2)5-12
execution, (2)5-12
formatted, (2)5-14
unformatted, (2)5-14
type, (1)2-1
and edit descriptors when SEGLDR and the EQUIV directive are used, (2)6-5
arithmetic expressions, (1)3-6
array element, (1)2-2
edit descriptors with, (2)6-5
function, (1)2-2
function subprogram, (1)2-2
DATA statement
definition, (2)2-4
features, E-10
format, (2)2-5
implied-DO list in a, (2)2-5
restrictions, (2)2-6
Dataset
connection, E-13
creation for random access, E-13
description, (2)5-3
identifier, format, (2)5-7
position, before data transfer, (2)5-5
unblocked, H-1
Date functions, B-5
DEBUG directive, (3)1-39
Debug package, symbolic, G-1
Debugging
aids, utility procedures, C-3
directives (DEBUG, NODEBUG), (3)1-39
Declaration statements
description, (2)2-1
double
description, E-10
FUNCTION, format, E-10
type, format, E-10
Declarator
array
actual, (1)2-11, (1)2-15
adjustable, (1)2-11
assumed-size, (1)2-11
description, (1)2-10
dummy, (1)2-11, (1)2-15
Declarator (continued)
format, (1)2-10
types, (1)2-11
dimension
description, (1)2-10
lower and upper bounds, (1)2-11
DECODE statement
description, E-9
format, E-7
Defined
entities, (1)2-21
initially, variable or array element, (1)2-21
variable or array element, (1)2-21
Deleting a file, (2)5-3
Dependencies, (3)2-4
Dependency messages, informative, D-30
Descriptor
edit
A, (2)6-21
and data types when SEGLDR and the EQUIV directive are used, (2)6-5
BN and BZ, (2)6-13
description, (2)6-3, E-9
dollar sign, (2)6-10.1
E, (2)6-16
F, (2)6-14
G, (2)6-17
H, (2)6-8
I, (2)6-13
L, (2)6-20
O (octal) (CFT extension), (2)6-19
obsolete, format, E-9
P, (2)6-11
R, (2)6-23
S, SP, and SS, (2)6-13
slash, (2)6-10
T, TL, TR, (2)6-8
with data types, (2)6-5
X, (2)6-8
Z (hexadecimal) (CFT extension), (2)6-20
nonrepeatable edit, format, (2)6-4
repeatable edit, format, (2)6-3
Design, entry block, F-1
Designator, substring
description, (1)2-18
format, (1)2-18
Deviations from the ANSI standards, (1)1-2
DFI CAL instruction, (3)2-15
Differences, named common and blank common, (2)2-15
Dimension
adjustable, (1)2-16
array, (1)2-10
declarator
description, (1)2-10
lower and upper bounds, (1)2-11
DIMENSION statement
description, (2)2-2
format, (2)2-2
Direct access
dataset position, (2)5-5
operations, (2)5-5

Direction of data transfer, (2)5-12
Directive
ALIGN
description, (3)1-37
format, (3)1-37
array bounds checking (BOUNDS), (3)1-34
BL, (3)1-33
BLOCK, (3)1-36
BOUNDS, (3)1-34
CODE
description, (3)1-23
format, (3)1-23
compiler
description, (3)1-20, (3)1-21
line, (3)1-21
control
integer (INT24, INT64), (3)1-29
listable output, (3)1-22
vectorization, (3)1-24
CVL, (3)1-39
DEBUG, (3)1-39
debugging, (3)1-39
DYNAMIC
description, (3)1-34
format, (3)1-34
dynamic common block (DYNAMIC)
description, (3)1-34
format, (3)1-34
EJECT
description, (3)1-22
format, (3)1-22
EQUIV, (2)6-5
FASTMD, (3)1-30
FLOW, (3)1-30
flow trace (FLOW/NOFLOW), (3)1-30
INT24
description, (3)1-29
format, (3)1-29
INT64
description, (3)1-29
format, (3)1-29
integer control (INT24, INT64)
description, (3)1-29
format, (3)1-29
IVDEP
description, (3)1-27
format, (3)1-27
IVDMO
description, (3)1-27
format, (3)1-27
line, compiler, (1)1-10, (3)1-21
LIST
description, (3)1-22
format, (3)1-23
listable output control, (3)1-22
multiply/divide (FASTMD, SLOWMD), (3)1-30
NEXTSCALAR
description, (3)1-28
format, (3)1-28
NOBL, (3)1-33
NOCODE
description, (3)1-23
format, (3)1-23
Directive (continued)
NOCVL, (3)1-39
NODEBUG, (3)1-39
NODOREP
description, (3)1-38
format, (3)1-38
NOFLOW, (3)1-30
NOIFCON
description, (3)1-37
format, (3)1-37
NOLIST
description, (3)1-23
format, (3)1-23
NORECURRENCE
description, (3)1-26
format, (3)1-26
NO SIDE EFFECTS
description, (3)1-36
format, (3)1-36
NOVECTOR
description, (3)1-25
format, (3)1-25
optimization, (3)1-35
RESUMEDOREP
description, (3)1-38
format, (3)1-38
RESUMEIFCON
description, (3)1-38
format, (3)1-38
ROLL, (3)1-39
SAFEIF, (3)1-33
scheduler, (3)1-33
SHORTLOOP
description, (3)1-29
format, (3)1-29
SLOWMD, (3)1-30
UNROLL, (3)1-39
UNSAFEIF, (3)1-33
VECTOR
description, (3)1-24
format, (3)1-24
vectorization control, (3)1-24
VFUNCTION
description, (3)1-27
format, (3)1-27
Disable, flow trace, (3)1-31
Disjunct, logical
form, (1)3-16
value, (1)3-16
Dollar sign
edit descriptor, (2)6-10.1
editing, (2)6-10.1
DO-loop
active, (2)4-11
description, (2)4-8
execution of the range, (2)4-13
extended range, E-15
inactive, (2)4-11
range, (2)4-10
transfer into range, (2)4-14
unrolling, (3)1-9
vectorizable, (3)2-1
DO statement
execution, (2)4-11
format, (2)4-8

DO statement (continued)
incrementation processing, (2)4-13
loop control processing, (2)4-13
range, (2)4-10
terminal statement
description, (2)4-10
execution, (2)4-13
DO variable, (2)4-10
Double declaration
FUNCTION statement, format, E-10
statement, E-10
type statement, format, E-10
Double-precision
constant
description, (1)2-6
range, nonzero, (1)2-6
data, (1)2-5
editing, (2)6-17
exponent, (1)2-6
expression, (1)3-6
type conversion, (1)3-9
DOUBLE PRECISION type statement, (2)2-7
Dummy
arguments
arrays, (1)4-12
association, (1)4-10
description, (1)4-9
procedures, (1)4-12
statement function, (2)7-3
type character, (1)4-11
undefined, (1)4-11
variables, (1)4-11
array
declarator, (1)2-11, (1)2-15
description, (1)2-15
procedure, symbolic name, (1)2-28
Dynamic common block directive (DYNAMIC)
description, (3)1-34
format, (3)1-34
DYNAMIC directive
description, (3)1-34
format, (3)1-34

E
edit descriptors, (2)6-16
editing, (2)6-16
Edit descriptor
A, (2)6-21
and data types when SEGLDR and the EQUIV directive are used, (2)6-5
BN and BZ, (2)6-13
description, (2)6-3, E-9
dollar sign, (2)6-10.1
E, (2)6-16
F, (2)6-14
G, (2)6-17
H, (2)6-8
hexadecimal (Z) (CFT extension), (2)6-20
I, (2)6-13
L (logical), (2)6-20
nonrepeatable, formats, (2)6-4
O (octal) (CFT extension), (2)6-19
obsolete, format, E-9
Edit descriptor (continued)
P, (2)6-11
R, (2)6-23
repeatable, formats, (2)6-3
S, SP, and SS, (2)6-13
slash, (2)6-10
T, TL, and TR, (2)6-8
with data types, (2)6-5
X, (2)6-8
Z (hexadecimal) (CFT extension), (2)6-20
Editing
A (alphanumeric), (2)6-21
apostrophe and quotation mark, (2)6-7
BN and BZ, (2)6-13
colon, (2)6-10
complex, (2)6-19
D, (2)6-17
data, (2)6-19
description, (2)5-1
dollar sign, (2)6-10.1
E, (2)6-16
F, (2)6-14
G, (2)6-17
H, (2)6-8
hexadecimal (Z) (CFT extension), (2)6-20
I, (2)6-13
integer, (2)6-13
L (logical), (2)6-20
numeric (BN, BZ, S, SP, SS, I, F, E, D, and G), (2)6-12
O (octal) (CFT extension), (2)6-19
P, (2)6-11
positional (T, TL, TR, and X), (2)6-8
R (right-justified) (CFT extension), (2)6-23
S, SP, and SS, (2)6-13
slash, (2)6-10
T, TL, and TR, (2)6-9
X, (2)6-9
Z (hexadecimal) (CFT extension), (2)6-20
Effect of ALLOC, SAVEALL, and BTREG on variable allocation, (3)1-10
Efficiency, vectorization, (3)2-11
EFI CAL instruction, (3)2-15
EJECT directive, format, (3)1-22
Element
array
arrangement and reference, (1)2-14
data type, (1)2-2
defined, (1)2-21
definition, (1)2-10
initially defined, (1)2-21
invariant, description, (3)2-2.1
invariant, used in a reduction array operation, (3)2-4
undefined, (1)2-21
name, array, (1)2-10, (1)2-12
order, array, (1)2-13
Elements of the CFT language, (1)1-3
ELSE-block, (2)4-7
ELSE IF-block, (2)4-6
ELSE IF statement
execution, (2)4-7
format, (2)4-7

ELSE statement, (2)4-7
EMA (extended memory addressing), (1)4-15
Empty sequence, (1)1-5
Enable/disable, flow trace, (3)1-31
ENCODE statement
    description, E-8
    format, E-7
Endfile record, (2)5-2
ENDFILE statement
    description, (2)5-17
    format, (2)5-16
END IF statement, (2)4-6
End-of-data (EOD) record, (2)5-2
End-of-file
    condition, (2)5-15
    (endfile) record, (2)5-2
END statement, (2)4-16
Entities
    associated
       partially, (1)2-19
       totally, (1)2-19
    association of
       description, (1)2-19
       restrictions, (1)4-13
    defined, (1)2-21
    global, (1)2-24
    local, (1)2-24
    undefined, (1)2-22
Entity categories, (3)2-2.1
Entry
    association in function subprograms,
      (2)7-9
    block design, F-1
    procedure subprogram, referencing,
      (2)7-9
ENTRY statement
    description, (2)7-8
    format, (2)7-8
    restrictions, (2)7-9
EOD, (2)5-2
EOF, IOEF, and IOSTAT functions, E-16
EQUIV directive, (2)6-5
Equivalence, mathematical, (1)3-10
EQUIVALENCE statement
    array
       element names, (2)2-12
       names, (2)2-12
    association, (2)2-12
    description, (2)2-11
    format, (2)2-11
    restrictions, (2)2-12, (2)2-15
Equivalenced arrays, (3)2-9
Error
    and end-of-file conditions, (2)5-15
    messages
       description, D-1
       during program execution, (3)1-11
       fatal, (1)4-15
    recovery, I/O, (2)5-34
    traceback, F-1
Establishing a format, (2)5-13
Evaluation
    arithmetic expressions, (1)3-10
    character expression, (1)3-11
Evaluation (continued)
    expressions, (1)3-19
    functions, order, (1)3-19
    subscript, (1)2-16
Executable
    program, (1)1-10, (1)1-11
    statement, (1)1-8
Execution
    arithmetic assignment statement, (2)3-1
    arithmetic IF statement, (2)4-4
    assigned GO TO statement, (2)4-3
    ASSIGN statement, (2)3-3
    block IF statement, (2)4-6
    CALL statement, (2)7-5
    character assignment statement, (2)3-2
    CLOSE statement, (2)5-21
    computed GO TO statement, (2)4-2
    conditional block statement, (2)4-8
    CONTINUE statement, (2)4-14
    data transfer I/O statement, (2)5-12
    DO-loop statement range, (2)4-13
    DO statement, (2)4-11
    ELSE IF statement, (2)4-7
    external function references, (1)4-5
    function references, (1)4-8
    INQUIRE by dataset name statement,
      (2)5-20
    INQUIRE by unit statement, (2)5-20
    logical
       assignment statement, (2)3-2
       IF statement, (2)4-5
    OPEN statement, (2)5-20
    program
       description, (1)1-12
       error messages, (3)1-11
    RETURN statement, (2)7-7
    sequence, normal, (1)1-12
    statement function reference, (1)4-4
    terminal statement, (2)4-13
    unconditional GO TO statement, (2)4-2
Exponent
    double-precision, (1)2-6
    real, (1)2-5
Exponential functions, B-3
Expressions
    arithmetic
       data type, (1)3-6
       description, (1)3-1
       evaluation, (1)3-10
       forms, (1)3-5
       relational, (1)3-12
    arithmetic constant, (1)3-1
    Boolean, (1)3-6
    Boolean (masking), (1)3-17
    character
       description, (1)3-11
       evaluation, (1)3-11
       relational, (1)3-13
    complex, (1)3-6
    description, (1)3-1
    double-precision, (1)3-6
    evaluation, (1)3-19
    Hollerith, E-4

Expressions (continued)
Hollerith relational
description, E-5
format, E-5
integer, (1)3-6
interpretation of arithmetic operators, (1)3-2
invariant, (3)2-2.1
logical
description, (1)3-14
form, (1)3-14, (1)3-16
interpretation, (1)3-15
value, (1)3-16
parenthesized, (1)3-20
real, (1)3-6
relational, (1)3-12
types, (1)3-1
Extended
memory
addressing (EMA), (1)4-15
common blocks, (1)4-15
range of a DO-loop, E-15
External
function, (1)4-5
actual arguments, (1)4-5
execution, (1)4-5
references, (1)4-5
symbolic name, (1)2-25
names, table, (3)1-19
routines, (3)1-40
unit
identifier, (2)5-6
specifier, format (2)5-18
EXTERNAL statement, format, (2)7-10

F
edit descriptor, (2)6-14
editing, (2)6-14
Factor
description, (1)3-4
logical
form, (1)3-15
value, (1)3-16
scale, (2)6-11
FASTMD directive, (3)1-30
Fatal error message, (1)4-15
Features
DATA statement, E-10
outmoded, E-1
Field
address, (3)1-16
block, (3)1-17
definition, (2)6-7
main usage, (3)1-17
name, (3)1-16
type, (3)1-16
width, (2)6-7
File
creating, (2)5-3
deleting, (2)5-3
description, (2)5-2
identifier, internal, (2)5-6
input/output, (2)5-2
File (continued)
internal
description, (2)5-4
restrictions, (2)5-4
positions, (2)5-3
specifier, format, (2)5-18
FLODUMP utility, (3)1-31
FLOW directive, (3)1-30
FLOWLIM flow trace routine option, (3)1-33
FLOW/NOFLOW directives, (3)1-30
Flow trace
directives (FLOW/NOFLOW), (3)1-30
enable/disable, (3)1-31
routine options, (3)1-32
Form
arithmetic expression, (1)3-5
logical
disjunct, (1)3-16
expression, (1)3-14, (1)3-16
factor, (1)3-15
term, (1)3-15
Format
ALIGN directive, (3)1-37
arithmetic assignment statement, (2)3-1
arithmetic IF statement, (2)4-4
array
declarator, (1)2-10
element name, (1)2-12
assigned GO TO statement, (2)4-3
ASSIGN statement, (2)3-3
BACKSPACE statement, (2)5-16
BLOCK DATA statement, (2)7-11
block IF statement, (2)4-6
BUFFER IN (CFT extension), (2)5-31
BUFFER OUT (CFT extension), (2)5-31
CALL ENDRPV, I-2
CALL SETRPV, I-1
CALL statement, (2)7-5
CFT control statement, (3)1-1
character assignment statement, (2)3-2
CHARACTER type statement, (2)2-8
CLOSE statement, (2)5-21
CODE directive, (3)1-23
COMMON statement, (2)2-13
computed GO TO statement, (2)4-2
CONTINUE statement, (2)4-14
control
information list (*cilist*), (2)5-9
positioning, (2)6-7
dataset identifier, (2)5-7
DATA statement, (2)2-5
DECODE statement, E-7
DIMENSION statement, (2)2-2
DO statement, (2)4-8
double declaration
FUNCTION statement, E-10
type statement, E-10
dynamic common block directive (DYNAMIC), (3)1-34
DYNAMIC directive, (3)1-34
EJECT directive, (3)1-22
ELSE IF statement, (2)4-7
ELSE statement, (2)4-7
ENCODE statement, E-7

Format (continued)
END statement, (2)4-16
ENDFILE statement, (2)5-16
END IF statement, (2)4-6
ENTRY statement, (2)7-8
EQUIVALENCE statement, (2)2-11
establishing a, (2)5-13
EXTERNAL statement, (2)7-10
external unit specifier, (2)5-18
file specifier, (2)5-18
format
identifier, (2)5-7
specification, (2)6-2
FORMAT statement, (2)6-1
function reference, (2)7-2
FUNCTION statement, (2)7-2
Hollerith relational expression, E-5
identifier, (2)5-7
IMPLICIT NONE statement (CFT extension) (2)2-10
IMPLICIT statement, (2)2-9
implied-DO list, (2)5-11
indirect logical IF statement, E-6
input NAMELIST group record, (2)5-24
INQUIRE by dataset name statement, (2)5-18
INQUIRE by unit statement, (2)5-18
INT24 directive, (3)1-29
INT64 directive, (3)1-29
integer control directives (INT24, INT64), (3)1-29
INTRINSIC statement, (2)2-15
I/O, list-directed, (2)6-24
IVDEP directive, (3)1-27
IVDMO directive, (3)1-27
list-directed I/O, (2)6-24
LIST directive, (3)1-23
logical
assignment statement, (2)3-2
IF statement, (2)4-4
NAMELIST statement, (2)5-23
NEXTSCALAR directive, (3)1-28
NOCODE directive, (3)1-23
NODOREP directive, (3)1-38
NOIFCON directive, (3)1-37
NOLIST directive, (3)1-23
nonrepeatable edit descriptors, (2)6-4
NORECURRENCE directive, (3)1-26
NO SIDE EFFECTS directive, (3)1-36
NOVECTOR directive, (3)1-25
obsolete edit descriptors, E-9
OPEN statement, (2)5-20
output NAMELIST group record, (2)5-28
PARAMETER statement, (2)2-1
PAUSE statement, (2)4-15
POINTER statement, (2)2-3
PRINT statement, (2)5-8
PROGRAM statement, (2)7-1
PUNCH statement, E-11
READ statement, (2)5-8
repeatable edit descriptors, (2)6-3
reprieve
initiation, I-1
termination, I-2

Format (continued)
RESUMEDOREP directive, (3)1-38
RESUMEIFCON directive, (3)1-38
RETURN statement, (2)7-6, (2)7-7
REWIND statement, (2)5-16
SAVE statement, (2)2-16
SHORTLOOP directive, (3)1-29
specification
description, (2)6-1
format, (2)6-2
Hollerith, E-6
interaction with I/O lists, (2)6-6
noncharacter arrays, E-15
statement function definition statement, (2)7-3
STOP statement, (2)4-15
SUBROUTINE statement, (2)7-6
substring designator, (1)2-18
task common block, (1)4-15
two-branch arithmetic IF statement, E-6
type statement, (2)2-7
type statements with data length, E-11
unconditional GO TO statement, (2)4-1
unit identifier, (2)5-6
VECTOR directive, (3)1-24
VFUNCTION directive, (3)1-27
WRITE statement, (2)5-8
FORMAT statement, (2)6-1
Formatted
data
assignment, E-7
transfer, (2)5-14
records, (2)5-1
FORTRAN
character set, (1)1-3, A-1
compiler, (1)1-1, (3)1-1, (3)2-1, D-1
intrinsic functions, B-1
language
definition, (1)1-1
elements, (1)1-3
specifications, (1)1-1
syntactic items, (1)1-5
programming, (3)2-1
statement, description, (1)1-8, (2)1-1
utility procedures, C-1
FORTRAN 77, (1)1-1
FTREF utility, J-1
Function
Boolean, B-5
character, B-4
data type, (1)2-2
definition statement format, statement, (2)7-3
dummy argument, statement, (2)7-3
EOF, E-16
execution of references, (1)4-7
exponential, B-3
external
actual arguments, (1)4-5
description, (1)4-5
execution, references, (1)4-5
referencing, (1)4-5
symbolic name, (1)2-25
general arithmetic, B-2

Function (continued)
IEOF, E-16
intrinsic
description, (1)4-6, (1)2-2, B-1
names, (2)2-16, B-1
referencing, (1)4-6
restrictions, (1)4-6
symbolic name, (1)2-28
IOSTAT, E-16
LENGTH (CFT extension), (2)5-34
logarithmic, B-3
maximum/minimum, B-4
miscellaneous, B-6
name, intrinsic, (2)2-16
order of evaluation, (1)3-19
reference, format, (2)7-2
references, execution, (1)4-8
referencing, (1)4-4, (1)4-8
restrictions, (1)4-7
statement
description, (1)4-3
execution, (1)4-4
referencing, (1)4-4
restrictions, (1)4-4
symbolic name, (1)2-28
subprogram
data type, (1)2-2
description, (1)4-3, (2)7-2
entry association, (2)7-9
restrictions, (1)4-7
time and date, B-5
trigonometric, B-3
type conversion, B-4
UNIT (CFT extension), (2)5-33
utility
LENGTH (CFT extension), (2)5-34
UNIT (CFT extension), (2)5-33
values, F-1
FUNCTION statement
double declaration, format, E-10
format, (2)7-2

G
edit descriptors, (2)6-17
editing, (2)6-17
General
arithmetic functions, B-2
guidelines for vectorization, (3)2-16
Generic and specific intrinsic function names, B-7
GETPOS/SETPOS, E-13
Global entities, (1)2-24
GO TO statement
assigned
execution, (2)4-3
format, (2)4-3
computed
execution, (2)4-2
format, (2)4-2
unconditional
execution, (2)4-2
format, (2)4-1
Group
name, NAMELIST, (1)2-29
record
input NAMELIST, format, (2)5-24
output NAMELIST, format, (2)5-28
Guidelines for vectorization, (3)2-16

H
edit descriptor, (2)6-8
editing, (2)6-8
Header lines, page, (3)1-14
Hexadecimal
Boolean, constant, (1)2-8
(Z) editing, (2)6-20
High-level language, (1)1-1
Hollerith
constants, E-2
expression, E-4
format specification, E-6
relational expression
description, E-5
format, E-5

I
edit descriptors, (2)6-13
editing, (2)6-13
Identifiers
dataset, format, (2)5-7
description, (2)5-6
format, format, (2)5-7
internal file, (2)5-6
unit
external, (2)5-6
format, (2)5-6
Identifying a unit, (2)5-13
IEOF function, E-16
IF-block, (2)4-5
IF-levels and blocks, (2)4-9
IF statements
and vectorization, (3)2-2
arithmetic
execution, (2)4-4
format, (2)4-4
block
execution, (2)4-6
format, (2)4-6
END, format, (2)4-6
indirect logical
description, E-6
format, E-6
logical
execution, (2)4-5
format, (2)4-4
two-branch arithmetic
description, E-6
format, E-6
IMPLICIT NONE statement (CFT extension)
description, (2)2-10
format, (2)2-10
IMPLICIT statement
description, (2)2-9
format, (2)2-9

Implied-DO list
description, (2)5-11
format, (2)5-11
in a DATA statement, (2)2-5
range, (2)2-6, (2)5-12
Inactive DO-loop, (2)4-11
Incrementation
parameter, (2)4-11
processing, (2)4-13
Increment integer, constant (CII), (3)2-2.1
Index, compressed, references, (3)2-16
Indirect logical IF statement
description, E-6
format, E-6
Information
cross-reference, (3)1-19
list, control (*cilist*), (2)5-9
Informative dependency messages, D-30
Initial
line, (1)1-9
parameter, (2)4-11
Initialization statements, (2)2-1
Initially defined variable or array element, (1)2-21
Initiation, reprieve
description, I-1
format, I-1
Input
list-directed, (2)6-24
list item, (2)5-10
NAMELIST
description, (2)5-24
group record, format, (2)5-24
physical record, (2)5-24
operations, random, E-13
processing, NAMELIST, (2)5-26
statement, (2)5-1
to CFT, (3)1-11
variables, NAMELIST, (2)5-25
Input/output
error recovery, (2)5-34
file, (2)5-2
list (*iolist*), (2)5-10
list-directed, (2)6-24
operations, random, E-13
records, (2)5-1
statements
description, (2)5-1
restrictions, (2)5-34
utility procedures, C-2, C-3
INQUIRE by dataset name statement
execution, (2)5-20
format, (2)5-18
INQUIRE by unit statement
execution, (2)5-20
format, (2)5-18
INQUIRE statement
description, (2)5-17
restrictions, (2)5-20
Inquiry specifiers and their meanings, (2)5-19
Instructions
CAL, EFI and DFI, (3)2-15
CMR (complete memory references), (3)2-18, (3)2-19
INT24 directive
description, (3)1-29
format, (3)1-29
INT64 directive
description, (3)1-29
format, (3)1-29
Integer
constant, (1)2-3, (1)2-4
constant increment (CII), (3)2-2.1
control directives (INT24, INT64)
description, (3)1-29
format, (3)1-29
data, (1)2-3
data item, (1)2-3
editing, (2)6-13
expression, (1)3-6
length conversion, B-4
quotient, (1)3-8
type conversion, (1)3-9
INTEGER type statement, (2)2-7
Interaction between I/O lists and format specification, (2)6-6
Internal
code, ASCII, (1)1-3
file
description, (2)5-4
identifier, (2)5-6
restrictions, (2)5-4
records, (2)5-4
representation, (2)6-7
Interpretation
arithmetic operators in expressions, (1)3-2
logical expression, (1)3-14
rules, (1)3-20
Intrinsic function
data type, (1)2-2
definition, (1)4-6
description, B-1
examples, (1)4-6
names, (2)2-16, B-7
referencing, (1)4-6
restrictions, (1)4-6
symbolic name, (1)2-28
INTRINSIC statement
description, (2)2-15
format, (2)2-15
Invariant
array element
description, (3)2-2.1
used in a reduction array operation, (3)2-4
description, (3)2-2.1
expression, (3)2-2.1
I/O
error recovery, (2)5-34
list-directed
description, (2)6-24
format, (2)6-24
lists, interaction with format specifications, (2)6-6
iolist (input/output list), (2)5-10
IOSTAT function, E-16

Items
    data, integer, (1)2-3
    list
       description, (1)1-7
       input, (2)5-10
       output, (2)5-11
    syntactic, (1)1-5, (1)1-7
Iteration count, (2)4-12
IVDEP directive
    description, (3)1-27
    format, (3)1-27
IVDMO directive
    description, (3)1-27
    format, (3)1-27

Keyword, (1)1-7

L
    edit descriptor, (2)6-20
    (logical) editing, (2)6-20
Label, statement, (1)1-6
Language
    CFT
       definition, (1)1-1
       elements, (1)1-3
    Cray FORTRAN, (1)1-1
    high-level, (1)1-1
len (length specification), (2)2-8, (2)6-22
Length
    data
       type statement, E-11
       type statements with, format, E-11
       with data types, E-12
    specification (*len*), (2)2-8, (2)6-22
    word, octal, with block names, (3)1-18
LENGTH function (CFT extension), (2)5-34
LENGTH, utility function, (2)5-34
Levels of messages, (3)1-2
Library routines
    CLEARFI, (3)2-15
    SENSEFI, (3)2-15
    SETFI, (3)2-15
    SYMDEBUG, G-1
Lines
    comment, (1)1-9
    compiler directive, (1)1-10, (3)1-21
    continuation, (1)1-9
    description, (1)1-8
    initial, (1)1-9
    order, (1)1-13
    page header, (3)1-14
    terminal, (1)1-9, (1)1-10
List
    control information (*cilist*), (2)5-9
    description, (1)1-7
    implied-DO
       description, (2)5-11
       range, (2)2-6, (2)5-12
    input/output (*iolist*), (2)5-10
    I/O, interaction with format specifications, (2)6-6
List (continued)
    item
       description, (1)1-7
       input, (2)5-10
       output, (2)5-11
Listable output
    control directives, (3)1-22
    description, (3)1-14
List-directed
    input, (2)6-24
    I/O
       description, (2)6-24
       format, (2)6-24
    output, (2)6-26
LIST directive
    description, (3)1-22
    format, (3)1-23
Listings, source statement, (3)1-15
Load operation, (3)2-18, (3)2-19
Local
    entities, (1)2-24
    temporary variable reference, F-1
Logarithmic functions, B-3
Logfile messages, D-28
Logical
    assignment statement
       execution, (2)3-2
       format, (2)3-2
    constant
       description, (1)2-7
       representation, (1)2-7
    data, (1)2-7
    disjunct
       form, (1)3-16
       value, (1)3-16
    editing (L), (2)6-20
    expression
       description, (1)3-14
       form, (1)3-16
       form and interpretation, (1)3-14
       value,(1)3-16
    factor
       form, (1)3-15
       value, (1)3-16
    IF statement
       execution, (2)4-5
       format, (2)4-4
       indirect, format, E-6
    interpretation, (1)3-15
    operands, (1)3-15
    operators, (1)3-14
    primaries, (1)3-15
    term
       form, (1)3-15
       value, (1)3-16
Logical IF statement, (2)4-4
LOGICAL type statement, (2)2-7
Loop control processing, (2)4-13
Loops
    encountered, table, (3)1-19
    vector, conditional, (3)2-10
    vectorizable, (3)2-17
Lower and upper bounds of dimension declarators, (1)2-11
Lowercase conversion, (1)1-5

Main
program
description, (1)1-12, (2)7-1
symbolic name, (1)2-26
usage field, (3)1-17
Mainframe
characteristics, (3)1-8
type, (3)1-8
Masking expression, Boolean, (1)3-17
Mathematical equivalence, (1)3-10
Maximum/minimum functions, B-4
Memory
allocation, (3)1-7
bidirectional, (3)2-17
extended
addressing (EMA), (1)4-15
common blocks, (1)4-15
Messages
BLOCK BEGINS, (3)1-15
CFT, D-1
compile-time, D-2
description, (3)1-20
error
description, D-1
during program execution, (3)1-11
fatal, (1)4-15
informative dependency, D-30
levels, (3)1-2
logfile, D-28
non-ANSI, (3)1-9
Miscellaneous functions, B-6
Modes
scalar, (3)2-4.1
vector, (3)2-4.1
Modifying a record under random access, E-15
Multiple stores, (3)2-18
Multiply/divide directives (FASTMD, SLOWMD), (3)1-30

Name
array
definition, (1)2-10
in an EQUIVALENCE statement, (2)2-12
use of, (1)2-17
array element
definition, (1)2-10, (1)2-12
in an EQUIVALENCE statement, (2)2-12
block, table, (3)1-18
common block, (1)2-25
external, table, (3)1-19
field, (3)1-16
group, NAMELIST, (1)2-29
intrinsic function, (2)2-16, B-1, B-7
symbolic
array, (1)2-26
block data subprogram, (1)2-26
classes, (1)2-25
common block, (1)2-25
constant, (1)2-27
description, (1)2-23
dummy procedure, (1)2-28
external function, (1)2-25
intrinsic function, (1)2-28
Name (continued)
main program name, (1)2-26
scope, (1)2-24
statement function, (1)2-28
subroutine, (1)2-26
variable, (1)2-27
variable, (1)2-27
Named common blocks, (1)4-1, (2)2-15
NAMELIST
group name, (1)2-29
input
description, (2)5-24
group record, format, (2)5-24
physical record, (2)5-24
processing, (2)5-26
variables, (2)5-25
output
description, (2)5-28
group record, format, (2)5-28
statement (CFT extension), format, (2)5-23
Names, symbolic
classes, (1)2-26
description, (1)1-6, (1)2-25
scope, (1)2-25
Names encountered, table, (3)1-16
NEXTSCALAR directive
description, (3)1-28
format, (3)1-28
NOBL directive, (3)1-33
NOCODE directive
description, (3)1-23
format, (3)1-23
NOCVL directive, (3)1-39
NODEBUG directive, (3)1-39
NODOREP directive
description, (3)1-38
format, (3)1-38
NOFLOW directive, (3)1-30
NOIFCON directive
description, (3)1-37
format, (3)1-37
NOLIST directive
description, (3)1-23
format, (3)1-23
Non-ANSI
features, outmoded, E-1
messages, (3)1-9
Noncharacter arrays for format specification, E-15
Non-executable statement, (1)1-8
Non-FORTRAN
procedures, creating, F-1
subprograms
creation of, F-1
description, (1)4-9
programming, F-1
Nonrepeatable edit descriptors, formats, (2)6-4
Nonzero
complex constant
description, (1)2-7
range, (1)2-7

Nonzero (continued)
double-precision constant
description, (1)2-6
range, (1)2-6
real constant
description, (1)2-5
range, (1)2-5
NORECURRENCE directive
description, (3)1-26
format, (3)1-26
Normal execution sequence, (1)1-12
NO SIDE EFFECTS directive
description, (3)1-36
format, (3)1-36
Notation
syntax of FORTRAN statement forms, (1)1-2
NOVECTOR directive
description, (3)1-25
format, (3)1-25
Numbers, table of statement, (3)1-15
Numeric editing, (BN, BZ, S, SP, SS, I, F, E, D, and G), (2)6-12

O (octal)
edit descriptor, (2)6-19
editing (CFT extension), (2)6-19
Obsolete edit descriptors, format, E-9
Octal
Boolean, constant, (1)2-8
(O) editing (CFT extension), (2)6-19
table of block names and lengths, (3)1-18
OPEN
specifiers and their meanings, (2)5-22
statement
execution, (2)5-20
format, (2)5-20
Operand
arithmetic, (1)3-3
description, (1)3-1
logical, (1)3-15
Operations
chaining, (3)2-18
direct access, (2)5-5
load, (3)2-18, (3)2-19
random input/output, E-13
sequential access, (2)5-4
store, (3)2-18, (3)2-19
Operator
arithmetic
description, (1)3-1, (1)3-2
interpretation in expressions, (1)3-2
precedence, (1)3-3
description, (1)1-7
logical, (1)3-14
precedence, (1)3-18
relational, (1)3-12
Optimization
directives, (3)1-35
options, (3)1-3
Optimized routines, (3)2-11
Optionally signed constant, (1)2-3

Options
ARGPLIMQ, (3)1-32
BOUNDS, (3)1-34
CFT Z, G-1
compiler, (3)1-12, (3)1-20
FLOWLIM, (3)1-33
optimization, (3)1-3
routine, flow trace, (3)1-32
SETPLIMQ, (3)1-32
Z, G-1
Order
element, array, (1)2-13
evaluation of functions, (1)3-19
lines, (1)1-13
statements, (1)1-13
Outmoded features, E-1
Output
control directives, listable, (3)1-22
from CFT, (3)1-14
listable, (3)1-14
list-directed, (2)6-26
list item, (2)5-11
NAMELIST, format, (2)5-28
operations, random, E-13
statement, (2)5-1

P
edit descriptor, (2)6-11
editing, (2)6-11
Package, symbolic debug, G-1
Page
header lines, (3)1-14
table, program unit, (3)1-20
Parameter
incrementation, (2)4-11
initial, (2)4-11
terminal, (2)4-11
PARAMETER statement
description, (2)2-1
format, (2)2-1
Parameters encountered, table, (3)1-18
Parentheses, (1)3-20
Parenthesized expression, (1)3-20
Partially associated entities, (1)2-19
PAUSE statement, format, (2)4-15
Physical record, input NAMELIST, (2)5-24
POINTER statement (CFT extension)
description, (2)2-3
format, (2)2-3
Position
dataset, before data transfer, (2)5-5
record and file, (2)5-3
Positional editing (T, TL, TR, and X), (2)6-8
Positioning
by format control, (2)6-7
while connected for random access (GETPOS/SETPOS), E-13
Precedence
arithmetic operators, (1)3-3
operators, (1)3-18

Primary
character, (1)3-11
description, (1)3-4
logical, (1)3-15
Print control characters, (2)5-15
Printing, (2)5-15
PRINT statement, format, (2)5-8
Procedures
dummy argument, (1)4-12
dummy, symbolic name, (1)2-28
non-FORTRAN, creating, F-1
utility, (1)4-6, C-1
Procedure subprograms
description, (1)4-2
referencing, (2)7-9
Processing
incrementation, (2)4-13
input, NAMELIST, (2)5-26
loop control, (2)4-13
reprieve, I-1
Program
control statements, (2)4-1
executable, (1)1-10, (1)1-11
execution
description, (1)1-12
error messages, (3)1-11
main, (1)1-12, (2)7-1
name, main, (1)2-26
subprogram, (1)1-10, (1)1-12, (2)7-1
unit
description, (1)1-10, (2)7-1
page table, (3)1-20
specification, (2)7-1
Programming, Cray FORTRAN, (3)2-1
PROGRAM statement, format, (2)7-1
Pseudo vector, (3)2-4
PUNCH statement
description, E-11
format, E-11

Qualifications for vectorization, (3)2-1
Quotation mark editing, (2)6-7
Quotient, integer, (1)3-8

R
edit descriptor, (2)6-23
editing (right-justified) (CFT extension), (2)6-23
Random
access
creating a dataset for, E-13
modifying a record under, E-15
positioning while connected for, E-13
input/output operations, E-13
Range
DO-loop
description, (2)4-10
execution, (2)4-13
extended, E-15
transfer into, (2)4-14
implied-DO list, (2)2-6, (2)5-12
Range (continued)
nonzero
complex constant, (1)2-7
double-precision constant, (1)2-6
real constant, (1)2-5
READ statement, format, (2)5-8
Reading, (2)5-1
READMS/WRITMS routines, E-14
Real
constant
basic, (1)2-5
description, (1)2-4
nonzero, (1)2-5
data, (1)2-4
exponent, (1)2-5
expression, (1)3-6
type conversion, (1)3-9
REAL type statement, (2)2-7
Record
description, (2)5-1
end-of-data, (2)5-2
end-of-file (endfile), (2)5-2
formatted, (2)5-1
group
input NAMELIST, format, (2)5-24
output NAMELIST, format, (2)5-28
input/output, (2)5-1
internal, (2)5-4
modification under random access, E-15
physical, input NAMELIST, (2)5-24
positions, (2)5-3
unformatted, (2)5-2
Recovery, I/O error, (2)5-34
Reference
array element, (1)2-14
compressed index, (3)2-16
description, (1)2-1
external function, execution, (1)4-5
function
execution, (1)4-8
format, (2)7-2
local temporary variable, F-1
statement function, execution, (1)4-4
subroutine
actual arguments, (1)4-2
description, (2)7-5
vector array, (3)2-3
Referencing
external functions, (1)4-5
functions, (1)4-8
intrinsic functions, (1)4-6
procedure subprogram entry, (2)7-9
statement functions, (1)4-4
Register use
A, S, V, VL, and VM, F-1
B and T, F-1
Relational
expressions
arithmetic, (1)3-12
character, (1)3-13
description, (1)3-12
Hollerith, format, E-5
operators, (1)3-12
Repeat specification, (2)6-3

Repeatable edit descriptor, (2)6-3
Replacement statement, conditional, (3)2-13
Representation
    internal, (2)6-7
    logical constant, (1)2-7
Reprieve
    initiation
       description, I-1
       format, I-1
    processing, I-1
    termination
       description, I-2
       format, I-2
Required order of lines and statements, (1)1-14
Restrictions
    association of entities, (1)4-13
    COMMON statement, (2)2-15
    DATA statement, (2)2-6
    ENTRY statement, (2)7-9
    EQUIVALENCE statement, (2)2-12, (2)2-15
    function subprogram, (1)4-7
    input/output statements, (2)5-34
    INQUIRE statement, (2)5-20
    internal file, (2)5-4
    intrinsic function, (1)4-6
    statement functions, (1)4-4
    subroutine subprogram, (1)4-3
RESUMEDOREP directive
    description, (3)1-38
    format, (3)1-38
RESUMEIFCON directive
    description, (3)1-38
    format, (3)1-38
Retrieval, argument, F-1
Return, alternate, (2)7-7
RETURN statement
    description, (2)7-6
    execution, (2)7-7
    format, (2)7-6, (2)7-7
REWIND statement
    description, (2)5-17
    format, (2)5-16
Right-justified (R) editing (CFT extension), (2)6-23
ROLL/UNROLL directives, (3)1-39
Routines
    external, (3)1-40
    library
       CLEARFI, (3)2-15
       SENSEFI, (3)2-15
       SETFI, (3)2-15
       SYMDEBUG, G-1
    optimized, (3)2-11
    options, flow trace, (3)1-32
    READMS/WRITMS, E-14
Rules of interpretation, (1)3-20

SAFEIF directive, (3)1-33
SAVE statement
    description, (2)2-16
    format, (2)2-16
Scalar
    mode, (3)2-4.1
    temporary, (3)2-3
Scale factor, (2)6-11
Scheduler directives, (3)1-33
Scope of symbolic name, (1)2-24
SEGLDR, (2)6-5
SENSEFI library routine, (3)2-15
Sequence
    collating, (1)1-3
    definition, (1)1-5
    empty, (1)1-5
    normal execution, (1)1-12
    storage
       array, (1)2-12.1, (1)2-13
       associated, (1)2-19
       common block, (2)2-14
       description, (1)2-19
       size, (1)2-19
Sequential access
    dataset positions, (2)5-5
    operations, (2)5-4
SETFI library routine, (3)2-15
SETPLIMQ flow trace routine option, (3)1-32
SETPOS, E-13
Sets, character
    ANSI FORTRAN, A-1
    ASCII, A-1
    Cray FORTRAN, A-1
    description, (1)1-3, A-1
SHORTLOOP directive
    description, (3)1-29
    format, (3)1-29
Signed constant, (1)2-3
Size
    array, (1)2-12
    common block, (2)2-14
    storage sequence, (1)2-19
Slash
    edit descriptor, (2)6-10
    editing, (2)6-10
SLOWMD directive, (3)1-30
Source
    code, (3)1-11
    statement listing, (3)1-15
Space Tables
    stack, (3)1-18
    static, (3)1-18
Special characters, (1)1-4
Specification
    data, (2)2-1
    format
       description, (2)6-1
       format, (2)6-2
       Hollerith, E-6
       interaction with I/O lists, (2)6-6
       noncharacter arrays for, E-15
    length (*len*), (2)2-8
    program unit, (2)7-1
    repeat, (2)6-3
    subprogram, (1)4-1

Specifier
CLOSE, and their meanings, (2)5-23
external unit, format, (2)5-18
file, format, (2)5-18
inquiry and meanings, (2)5-19
OPEN, and their meanings, (2)5-22
S, SP, and SS
edit descriptors, (2)6-13
editing, (2)6-13
Stack Space Table, (3)1-18
Standard, ANSI
conformance with, (1)1-1
deviations from, (1)1-2
Statement
arithmetic assignment
execution, (2)3-1
format, (2)3-1
arithmetic IF
execution, (2)4-4
format, (2)4-4
ASSIGN
execution, (2)3-3
format, (2)3-3
assigned GO TO
execution, (2)4-3
format, (2)4-3
assignment
description, (2)3-1
type conversion, (1)3-8
association, (2)2-11
BACKSPACE
description, (2)5-17
format, (2)5-16
BLOCK DATA, format, (2)7-11
block IF
execution, (2)4-6
format, (2)4-6
BUFFER IN (CFT extension)
description, (2)5-29
format, (2)5-31
BUFFER OUT (CFT extension)
description, (2)5-29
format, (2)5-31
CALL
execution, (2)7-5
format, (2)7-5
CFT control, format, (3)1-1
character assignment
execution, (2)3-2
format, (2)3-2
CLOSE
execution, (2)5-21
format, (2)5-21
COMMON
description, (2)2-13
format, (2)2-13
restrictions, (2)2-15
computed GO TO
execution, (2)4-2
format, (2)4-2
conditional, (3)2-13
conditional block
description, (2)4-5
execution, (2)4-8
Statement (continued)
conditional replacement, (3)2-13
CONTINUE
execution, (2)4-14
format, (2)4-14
DATA
description, (2)2-4
features, E-10
format, (2)2-5
restrictions, (2)2-6
data specification, (2)2-1
declaration, (2)2-1
DECODE
description, E-9
format, E-7
DIMENSION
description, (2)2-2
format, (2)2-2
DO
execution, (2)4-11
format, (2)4-8
double declaration
description, E-10
FUNCTION, format, E-10
type, format, E-10
ELSE, (2)4-7
ELSE-block, (2)4-7
ELSE IF
execution, (2)4-7
format, (2)4-7
ELSE IF-block, (2)4-6
ENCODE
description, E-8
format, E-7
END, format, (2)4-16
ENDFILE
description, (2)5-17
format, (2)5-16
END IF, format, (2)4-6
ENTRY
description, (2)7-8
format, (2)7-8
restrictions, (2)7-9
EQUIVALENCE
description, (2)2-11
format, (2)2-11
restrictions, (2)2-12, (2)2-15
executable, (1)1-8
EXTERNAL, format, (2)7-10
FORMAT, format, (2)6-1
FORTRAN, (1)1-8, (2)1-1
FUNCTION, format, (2)7-2
IF-block, (2)4-5
IMPLICIT
description, (2)2-9
format, (2)2-9
IMPLICIT NONE (CFT extension)
description, (2)2-10
format, (2)2-10
indirect logical IF
description, E-6
format, E-6
initialization, (2)2-1

Statement (continued)
input/output
description, (2)5-1
restrictions, (2)5-34
INQUIRE
description, (2)5-17
restrictions, (2)5-20
INQUIRE by dataset name
execution, (2)5-20
format, (2)5-18
INQUIRE by unit
execution, (2)5-20
format, (2)5-18
INTRINSIC
description, (2)2-15
format, (2)2-15
label, (1)1-6
logical assignment
execution, (2)3-2
format, (2)3-2
logical IF
execution, (2)4-5
format, (2)4-4
NAMELIST (CFT extension), format, (2)5-23
non-executable, (1)1-8
OPEN
execution, (2)5-20
format, (2)5-20
order, (1)1-13
PARAMETER
description, (2)2-1
format, (2)2-1
PAUSE, format, (2)4-15
POINTER (CFT extension)
description, (2)2-2
format, (2)2-2
PRINT, format, (2)5-8
PROGRAM, format, (2)7-1
program control, (2)4-1
PUNCH
description, E-11
format, E-11
READ, format, (2)5-8
RETURN
description, (2)7-6
execution, (2)7-7
format, (2)7-6, (2)7-7
REWIND
description, (2)5-17
format, (2)5-16
SAVE
description, (2)2-16
format, (2)2-16
source, listings, (3)1-15
statement function definition, format, (2)7-3
STOP, format, (2)4-15
SUBROUTINE, format, (2)7-6
terminal
description, (2)4-10
execution, (2)4-13
two-branch arithmetic IF
description, E-6
format, E-6

Statement (continued)
type
CHARACTER, (2)2-8
COMPLEX, (2)2-7
data length, E-11
description, (2)2-7
double declaration, E-10
DOUBLE PRECISION, (2)2-7
format, (2)2-7
INTEGER, (2)2-7
LOGICAL, (2)2-7
REAL, (2)2-7
with data length, format, E-11
unconditional GO TO
execution, (2)4-2
format, (2)4-1
WRITE, format, (2)5-8
Statement function, (1)4-3
definition statement, format, (2)7-3
dummy argument, (2)7-3
execution, (1)4-4
referencing, (1)4-4
restrictions, (1)4-4
symbolic name, (1)2-28
Statement numbers, table, (3)1-15
Static calling tree, J-1
Static Space Table, (3)1-18
Status, vector, of subprograms, B-1
STOP statement, format, (2)4-15
Storage sequence
array, (1)2-12.1, (1)2-13
associated, (1)2-19
common block, (2)2-14
description, (1)2-19
size, (1)2-19
Store operation, (3)2-18, (3)2-19
Stores, multiple, (3)2-18
Subprogram
block data
description, (1)4-1, (2)7-11
symbolic name, (1)2-26
description, (1)1-12, (1)4-1, (2)7-1
entry, procedure
referencing, (2)7-9
function
data type, (1)2-2
description, (1)4-3, (2)7-2
entry association, (2)7-9
restrictions, (1)4-7
non-FORTRAN, (1)4-9
procedure, (1)4-2
specification, (1)4-1
subroutine
description, (1)4-2
restrictions, (1)4-3
types, (1)4-1
vector status of, B-1
Subroutine
actual arguments, (1)4-2
reference, (2)7-5
subprogram
description, (1)4-2
restrictions, (1)4-3
symbolic name, (1)2-26
user control, (2)5-26, (2)5-28

SUBROUTINE statement, format, (2)7-6
Subscript
    evaluation, (1)2-16
    values, (1)2-15
Substring
    character
        actual argument, (1)4-11
        description, (1)2-18
    designator
        description, (1)2-18
        format, (1)2-18
Summary, rules of interpretation, (1)3-20
Symbolic debug package, G-1
Symbolic names
    array, (1)2-26
    block data subprogram, (1)2-26
    classes, (1)2-25
    common block, (1)2-25
    constant, (1)2-27
    description, (1)1-6, (1)2-23
    dummy procedure, (1)2-28
    external function, (1)2-25
    intrinsic function, (1)2-28
    main program, (1)2-26
    scope, (1)2-24
    statement function, (1)2-28
    subroutine, (1)2-26
    variable, (1)2-27
SYMDEBUG library routine, G-1
Syntactic items, (1)1-5, (1)1-7
Syntax, FORTRAN statement forms, (1)1-2
System utility procedures, C-2

T, TL, and TR
    edit descriptors, (2)6-8
    editing, (2)6-9
Table
    block names and lengths in octal, (3)1-18
    external names, (3)1-19
    loops encountered, (3)1-19
    names encountered, (3)1-16
    parameters encountered, (3)1-18
    program unit page, (3)1-20
    Stack Space, (3)1-18
    statement numbers, (3)1-15
    Static Space (3)1-18
Task common blocks
    description, (1)4-15
    format, (1)4-15
Temporary
    scalar, (3)2-3
    variable reference, local, F-1
Term
    description, (1)3-5
    logical
        form, (1)3-15
        value, (1)3-16
Terminal
    line, (1)1-9, (1)1-10
    parameter, (2)4-11
    statement
        description, (2)4-10
        execution, (2)4-13
Termination, reprieve, I-2
Time and date functions, B-5
Time utility procedures, C-2
Totally associated entities, (1)2-19
Trace, flow
    directives, (3)1-30
    enable/disable, (3)1-31
    routine options, (3)1-32
Traceback, error, F-1
Transfer
    data
        dataset position before, (2)5-5
        description, (2)5-12, (2)5-13
        direction of, (2)5-12
        execution, (2)5-12
        formatted, (2)5-14
        unformatted, (2)5-14
    into range of a DO-loop, (2)4-14
Transmission, argument, F-1
Trigonometric functions, B-3
Two-branch arithmetic IF statement
    description, E-6
    format, E-6
Types
    array declarator, (1)2-11
    character, dummy argument, (1)4-11
    conversion
        assignment statements, (1)3-8
        Boolean, (1)3-10
        complex, (1)3-10
        description, (1)3-9
        double-precision, (1)3-9
        functions, B-4
        integer, (1)3-9
        real, (1)3-9
    data, (1)2-1
        and edit descriptors when SEGLDR and the EQUIV directive are used, (2)6-5
        array element, (1)2-2
        edit descriptors with, (2)6-5
        function, (1)2-2
        function subprogram, (1)2-2
        of arithmetic expression, (1)3-6
    field, (3)1-16
    mainframe, (3)1-8
    statement
        CHARACTER, (2)2-8
        COMPLEX, (2)2-7
        data length, E-11
        description, (2)2-7
        double declaration, format, E-10
        DOUBLE PRECISION, (2)2-7
        format, (2)2-7
        INTEGER, (2)2-7
        LOGICAL, (2)2-7
        REAL, (2)2-7
        with data length, format, E-11

Unblocked datasets, H-1
Unconditional GO TO statement
    execution, (2)4-2
    format, (2)4-1

Undefined
    dummy argument, (1)4-11
    entities, (1)2-22
    variable or array element, (1)2-21
Unformatted
    data transfer, (2)5-14
    records, (2)5-2
Unit
    description, (2)5-6
    identifier
       external, (2)5-6
       format, (2)5-6
    identifying, (2)5-13
    program
       description, (1)1-10, (2)7-1
       page table, (3)1-20
    specification, program (2)7-1
    specifier, external, (2)5-18
UNIT
    function (CFT extension), (2)5-33
    utility function, (2)5-33
UNROLL directive, (3)1-39
Unrolling, DO-loop, (3)1-9
UNSAFEIF directive, (3)1-33
Unsigned constant, (1)2-3
Upper bounds of dimension declarators, (1)2-11
Uppercase/lowercase conversion, (1)1-5
Use of
    array names, (1)2-17
    optimized routines by CFT, (3)2-12
User
    control subroutine argument lists, (2)5-27
    control subroutines, (2)5-26, (2)5-28
Using optimized routines, (3)2-11
Utility
    FLODUMP, (3)1-31
    FTREF, J-1
    function
       LENGTH, (2)5-34
       UNIT, (2)5-33
    procedures
       debugging aids, C-3
       description, (1)4-6, C-1
       input/output, C-2, C-3
       system, C-2
       time, C-2

Values
    function, F-1
    logical
       disjunct, (1)3-16
       expression, (1)3-16
       factor, (1)3-16
       term, (1)3-16
    subscript, (1)2-15
Value separators, (2)6-24
Variable
    defined, (1)2-21
    description, (1)2-9
    DO, (2)4-10
    dummy arguments, (1)4-11
Variable (continued)
    initially defined, (1)2-21
    name, (1)2-27
    NAMELIST input, (2)5-25
    reference, local temporary, F-1
    symbolic name, (1)2-27
    undefined, (1)2-21
    used in a reduction array operation, (3)2-4
Vector
    array reference, (3)2-3
    loops, conditional, (3)2-10
    mode, (3)2-4.1
    pseudo, (3)2-4
    status of subprograms, B-1
VECTOR directive
    description, (3)1-24
    format, (3)1-24
Vectorizable
    DO-loops, (3)2-1
    loops, (3)2-17
Vectorization
    aids, B-6
    and IF statements, (3)2-2
    conditions inhibiting, (3)2-2
    control directives, (3)1-24
    efficiency, (3)2-11
    guidelines, (3)2-16
    qualifications, (3)2-1
    with arrays, (3)2-10
VFUNCTION directive
    description, (3)1-27
    format, (3)1-27

Width, field, (2)6-7
WRITE statement, format, (2)5-8
Writing, (2)5-1
WRITMS routine, E-14

X
    edit descriptor, (2)6-8
    editing, (2)6-9

Z (hexadecimal)
    edit descriptor, (2)6-20
    editing (CFT extension), (2)6-20
Z option, CFT, G-1

READERS COMMENT FORM

FORTRAN (CFT) Reference Manual SR-0009 J-03

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

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

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.