CRAY® COMPUTER SYSTEMS

CFT77
REFERENCE MANUAL
SR-0018 Rev B

Copyright© 1986, 1988 by Cray Research, Inc. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

RECORD OF REVISION **CRAY RESEARCH, INC.** PUBLICATION NUMBER SR-0018

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

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

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

CRAY RESEARCH, INC.
1345 Northland Drive
Mendota Heights, Minnesota 55120

Revision	Description
	April 1986 - Original printing.
A	September 1986 - Changes are the SUPPRESS directive and the TARGET command. Sections on input/output have been reorganized, with a new introduction in section 7. Other editorial changes have been made. Trademarks are now documented in the record of revision. The previous version is obsolete.
B	February 1988 - This reprint with revision adds the INCLUDE statement, Loopmark feature, BL and NOBL directives, ALLOC directive, INTEGER directive, I/INDEF option, -V option (UNICOS only), EDN keyword (COS only), and P and w options (CRAY-2 systems only). Section 8, I/O formatting, is reorganized. Descriptions of compiling and the use of operating system features and utilities have been expanded. An appendix on dynamic memory management has been added.

The UNICOS operating system is derived from the AT&T UNIX System V operating system. UNICOS is also based, in part, on the Fourth Berkeley Software Distribution under license from The Regents of the University of California.

CRAY, CRAY-1, SSD, and UNICOS are registered trademarks and CFT, CFT77, CFT2, COS, CRAY-2, CRAY X-MP, CRAY Y-MP, CSIM, HSX, IOS, SEGLDR, and SUPERLINK are trademarks of Cray Research, Inc.

PREFACE

This is a reference manual for the Cray Fortran compiler CFT77, which operates on all Cray computers and operating systems. The manual includes a complete description of Fortran, CFT77 features, and instructions for using the compiler. An effort has been made to make this manual serve both as a Fortran text and as an introduction to the use of Cray computers, to reduce the need for consulting other manuals and books. This is accomplished as follows:

- The use of the Cray operating systems UNICOS and COS is shown for typical programming needs.
- Reference material includes coding issues and examples.
- Tutorials are included for the benefit of programmers who are not expert in Fortran, such as an introduction to I/O in 7.1.
- The use of Cray programming tools is introduced: debuggers (DEBUG, 1.1.3 and 1.2.4); program flow tracing (Flowtrace, 1.4.4.1); analysis of program structure and cross references (FTREF, 4.6.1); analysis of program performance (**prof** and SPY, 10.1.3); and machine performance monitoring (Perftrace, 10.1.4).

Change bars appear on some text to indicate the most significant changes from the previous edition of this manual (SR-0018 A). These do not necessarily indicate changes in the CFT77 compiler; many change bars appear on new discussions of existing usage issues. A change bar on the page footer indicates that most of that page has been changed.

Of the numerous books about Fortran, the following can be recommended:

Merchant, Michael J. *Fortran 77: Language and Style*. Belmont, CA: Wadsworth, 1981. Introductory text with emphasis on programming style.

Ellis, T.M.R. *A Structured Approach to Fortran 77 Programming*. Reading, MA: Addison-Wesley, 1981. Introductory text that explains the ANSI standard in detail.

Press, William H., et. al. *Numerical Recipes: The Art of Scientific Programming* and *Numerical Recipes: Example Book (Fortran)*. Explains the selection of algorithms for many mathematical purposes.

Metcalf, Michael. *Fortran Optimization*. New York: Academic Press, 1985. Discusses various approaches to improving execution speed.

Two reference cards are available for CFT77:

SQ-0138 UNICOS CFT77 Reference Card
SQ-0137 COS CFT77 Reference Card

The following publications describe the UNICOS operating system:

SG-2052 UNICOS Overview for Users
SG-2010 UNICOS Primer
SG-2050 UNICOS Text Editors Primer
SQ-2054 UNICOS *vi* Reference Card
SR-2040 UNICOS Performance Utilities Reference Manual
SR-2011 UNICOS User Commands Reference Manual
SR-2013 CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual
SR-0112 UNICOS Symbolic Debugging Package Reference Manual

The following publications describe Cray software that is not specific to an operating system:

SR-0113 Programmer's Library Reference Manual
SR-0066 Segment Loader (SEGLDR) Reference Manual

The following publications describe the COS operating system:

SR-0146 COS Performance Utilities Reference Manual
SR-0011 COS Reference Manual
SR-0012 Macros and Opdefs Reference Manual

CONTENTS

PREFACE iii

1. THE CFT77 COMPILER 1-1

1.1 USING CFT77 UNDER UNICOS 1-2
1.1.1 Compiling and running a program 1-2
1.1.2 Issues for successful use of CFT77 1-3
1.1.3 Preparing for debugging under UNICOS 1-4
1.1.4 The **cft77** command under UNICOS 1-5
1.1.5 Compiler options under UNICOS 1-10
1.2 USING CFT77 UNDER COS 1-13
1.2.1 Preparing your program as a COS job 1-13
1.2.2 Running your job under COS: operations and datasets 1-14
1.2.3 Using datasets that are specific to your job . . 1-14
1.2.4 Preparing for debugging under COS 1-16
1.2.5 Issues for successful use of CFT77 1-16
1.2.6 The CFT77 control statement under COS 1-17
1.2.7 Compiler options under COS 1-23
1.3 CROSS-COMPILING USING THE -C OR CPU= KEYWORD 1-25
1.4 COMPILER DIRECTIVES 1-27
1.4.1 Vectorization directives 1-28
1.4.1.1 Suppressing vectorization (VECTOR and NOVECTOR) 1-28
1.4.1.2 Ignore dependencies (IVDEP) 1-28
1.4.1.3 Vectorizable functions (VFUNCTION) . . 1-29
1.4.1.4 Loops with low trip counts (SHORTLOOP) 1-30
1.4.1.5 Register storage across subprograms (NO SIDE EFFECTS) 1-30
1.4.2 Scalar optimization directives 1-31
1.4.2.1 Momentary suppression (SUPPRESS) . . . 1-31
1.4.2.2 Bottom loading of operands (BL/NOBL) . 1-32
1.4.3 Output listing directives 1-32
1.4.3.1 Inserting a page break (EJECT) 1-33
1.4.3.2 Listing of source program (LIST and NOLIST) 1-33
1.4.3.3 Listing of generated code (CODE and NOCODE) 1-33

1.4 COMPILER DIRECTIVES (continued)
1.4.4 Localized control of command options 1-33
1.4.4.1 Flowtrace (FLOW and NOFLOW) 1-33
1.4.4.2 Array bounds checking (BOUNDS and NOBOUNDS) 1-34
1.4.4.3 Storage allocation (ALLOC) 1-35
1.4.4.4 Integer length (INTEGER) 1-35
1.4.5 Dynamic common block directive (DYNAMIC) 1-36
1.5 INCLUDE STATEMENT - INSERTING EXTERNAL SOURCE FILES . . 1-36
1.6 LISTABLE OUTPUT 1-37
1.7 CROSS-REFERENCE LISTINGS 1-38
1.7.1 Symbol Cross-reference Table 1-38
1.7.1.1 Name, address, and type fields 1-39
1.7.1.2 Usage field 1-39
1.7.1.3 Storage field 1-39
1.7.1.4 Source program references 1-40
1.7.2 Parameter Table 1-41
1.7.3 Label Cross-reference Table 1-41

2. LANGUAGE ELEMENTS AND STRUCTURE 2-1

2.1 ELEMENTS OF THE FORTRAN LANGUAGE 2-1
2.1.1 Character set 2-1
2.1.2 Syntactic items 2-2
2.1.3 Lines 2-2
2.1.3.1 Initial and terminal lines 2-3
2.1.3.2 Continuation lines 2-3
2.1.3.3 Comment lines and embedded comments . 2-3
2.1.3.4 Compiler directive lines 2-4
2.1.4 Statements 2-4
2.1.4.1 Kinds of statements 2-4
2.1.4.2 Order of statements and lines 2-5
2.1.5 Symbolic names 2-7
2.2 THE EXECUTABLE PROGRAM 2-8
2.2.1 Procedures: subroutines and functions 2-8
2.2.2 Summary of program structure 2-9
2.2.3 Communicating data within programs 2-11
2.3 PROGRAM UNITS 2-11
2.3.1 PROGRAM statement 2-12
2.4 FUNCTIONS 2-13
2.4.1 Function reference 2-13
2.4.1.1 Data type of a function: reference versus value 2-14
2.4.1.2 Execution of functions 2-15
2.4.1.3 Order of evaluation 2-16
2.4.2 Statement functions 2-16
2.4.2.1 Statement function definition statement 2-17
2.4.3 Intrinsic functions 2-19
2.4.3.1 Referencing intrinsic functions . . . 2-20
2.4.3.2 Restrictions 2-20

2. LANGUAGE ELEMENTS AND STRUCTURE (continued)

2.5 SUBPROGRAMS 2-21
2.5.1 External functions and function subprograms . . 2-22
2.5.1.1 Restrictions on external functions . . 2-22
2.5.1.2 Function subprograms 2-22
2.5.1.3 FUNCTION statement 2-24
2.5.2 Subroutines and subroutine subprograms 2-25
2.5.2.1 Requirements 2-26
2.5.2.2 CALL statement (subroutine reference) 2-26
2.5.2.3 SUBROUTINE statement 2-28
2.5.3 Altering the transfer of control between program units 2-28
2.5.3.1 ENTRY statement 2-29
2.5.3.2 RETURN statement 2-30
2.6 ARGUMENTS 2-32
2.6.1 Association of arguments 2-32
2.6.2 Actual arguments for external procedures 2-33
2.6.3 Dummy arguments 2-34
2.6.4 Dummy procedures 2-35
2.6.4.1 EXTERNAL statement 2-36
2.6.4.2 INTRINSIC statement 2-36

3. DATA TYPES 3-1

3.1 TYPE SPECIFICATION 3-3
3.1.1 Type statements 3-3
3.1.2 IMPLICIT statement 3-4
3.1.3 IMPLICIT NONE statement (CFT77 extension) . . . 3-6
3.2 INTEGER TYPE 3-6
3.3 REAL TYPE 3-7
3.4 DOUBLE-PRECISION TYPE 3-8
3.5 COMPLEX TYPE 3-9
3.6 LOGICAL TYPE 3-9
3.7 CHARACTER TYPE 3-10
3.7.1 CHARACTER type statement 3-11
3.7.1.1 Asterisk specification 3-11
3.7.1.2 Character function declaration 3-12
3.7.2 Character substrings 3-13
3.7.3 Arguments of type character 3-14
3.8 BOOLEAN TYPE (CFT77 EXTENSION) 3-15
3.9 POINTER TYPE (CFT77 EXTENSION) 3-16
3.9.1 POINTER type statement (CFT77 extension) 3-17
3.9.2 Using pointers 3-18

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

4.1 CONSTANTS 4-1
4.1.1 PARAMETER statement 4-1
4.2 VARIABLES 4-3

4. DATA STRUCTURES, STORAGE, AND ASSOCIATION (continued)

4.3 ARRAYS 4-4
4.3.1 Dummy, actual, and pointee arrays 4-5
4.3.2 Constant, adjustable, and assumed-size arrays . 4-5
4.3.3 Automatic arrays (CFT77 extension) 4-6
4.3.4 DIMENSION statement 4-6
4.3.5 Array declarators 4-7
4.3.5.1 Kinds of array declarators 4-8
4.3.6 Array elements and subscripts 4-9
4.3.6.1 Array subscripts and storage sequence 4-10
4.3.7 Array size 4-12
4.3.8 Arrays as arguments 4-12
4.3.9 Use of array names 4-13
4.3.10 Array section (CFT77 extension) 4-16
4.3.10.1 Uses and restrictions 4-16
4.3.10.2 Array section name 4-17
4.3.10.3 Indexed section selectors 4-17
4.3.10.4 Vector-valued section selectors . . . 4-18
4.3.11 Array expressions (CFT77 extension) 4-20
4.3.11.1 Conformance of array operands 4-21
4.3.11.2 Order of operations 4-21
4.3.11.3 Array operands in intrinsic functions 4-23
4.4 DATA STATEMENT 4-24
4.4.1 Implied-DO list in a DATA statement 4-26
4.4.2 Data types in a DATA statement 4-27
4.4.3 Entities that can appear in a DATA statement . . 4-28
4.5 STORAGE AND ASSOCIATION 4-28
4.5.1 Storage units and sequences 4-28
4.5.2 Static and stack storage 4-29
4.5.3 Definition 4-30
4.5.4 SAVE statement 4-31
4.5.5 Association of entities 4-32
4.5.5.1 Implicit association 4-33
4.5.5.1 Global and local data (Cray terminology) 4-34
4.5.6 EQUIVALENCE statement 4-34
4.5.6.1 Array names in EQUIVALENCE statements 4-36
4.5.6.2 Restrictions on EQUIVALENCE statements 4-36
4.6 COMMON BLOCKS 4-37
4.6.1 Features and utilities for using common blocks . 4-38
4.6.2 COMMON statement 4-39
4.6.3 Referencing common blocks 4-39
4.6.4 Common block storage sequence 4-40
4.6.5 Common block size 4-42
4.6.6 TASK COMMON statement (CFT77 extension) 4-43
4.6.7 LOCAL COMMON statement (for CRAY-2 systems) . . 4-44
4.6.8 Block data subprogram 4-44
4.8.8.1 BLOCK DATA statement 4-45

5. EXPRESSIONS AND ASSIGNMENT 5-1

5.1 ARITHMETIC EXPRESSIONS 5-2
5.1.1 Arithmetic assignment statement 5-3
5.1.2 Arithmetic operators 5-5
5.1.2.1 Precedence of arithmetic operators . . 5-5
5.1.3 Arithmetic operands 5-6
5.1.3.1 Primaries 5-6
5.1.3.2 Factors 5-7
5.1.3.3 Terms 5-7
5.1.3.4 Arithmetic expressions 5-8
5.1.4 Data type of arithmetic expressions 5-8
5.1.4.1 Type conversion 5-12
5.1.5 Considerations in evaluating arithmetic expressions 5-13
5.2 CHARACTER EXPRESSIONS 5-13
5.2.1 Character assignment statement 5-14
5.2.2 Character expression evaluation 5-15
5.2.3 Hollerith type 5-15
5.3 RELATIONAL EXPRESSIONS 5-15
5.3.1 Arithmetic relational expressions 5-16
5.3.2 Character relational expressions 5-17
5.4 LOGICAL EXPRESSIONS 5-17
5.4.1 Logical assignment statement 5-19
5.4.2 Logical operators 5-19
5.4.3 Form and interpretation of logical expressions . 5-21
5.5 MASKING EXPRESSIONS (CFT77 extension) 5-22

6. PROGRAM CONTROL 6-1

6.1 CONDITIONAL BLOCKS 6-1
6.1.1 Block IF statement 6-4
6.1.2 ENDIF statement 6-4
6.1.3 ELSEIF statement 6-4
6.1.4 ELSE statement 6-5
6.2 OTHER IF STATEMENTS 6-5
6.2.1 Logical IF statement 6-5
6.2.2 Arithmetic IF statement 6-6
6.3 DO LOOPS 6-6
6.3.1 DO statement 6-8
6.3.2 Terminal statement and CONTINUE statement . . . 6-9
6.3.3 Loop control and incrementation processing . . . 6-10
6.4 GOTO AND ASSIGN STATEMENTS 6-11
6.4.1 Unconditional GOTO statement 6-11
6.4.2 Computed GOTO statement 6-12
6.4.3 Assigned GOTO statement 6-12
6.4.4 ASSIGN statement 6-13
6.5 SUSPENDING AND HALTING EXECUTION 6-14
6.5.1 STOP statement 6-14
6.5.2 END statement 6-15
6.5.3 PAUSE statement 6-15

7. I/O OVERVIEW, TERMS, STATEMENTS 7-1

7.1 I/O TUTORIAL 7-1
7.1.1 Using nondefault files 7-1
7.1.1.1 Using a data file from the front end . 7-2
7.1.1.2 Creating a file for transfer to the front end 7-2
7.1.1.3 Creating a file for further processing 7-3
7.1.2 The three kinds of standard I/O 7-3
7.1.2.1 Unformatted I/O 7-4
7.1.2.2 List-directed I/O 7-4
7.1.2.3 Formatted I/O 7-5
7.1.3 Examples of formatted I/O 7-6
7.1.4 Increasing I/O performance 7-8
7.2 INPUT/OUTPUT RECORDS 7-10
7.3 INPUT/OUTPUT FILES AND DATASETS 7-11
7.3.1 File structures 7-12
7.3.2 File identifier 7-13
7.3.3 COS dataset 7-14
7.3.3.1 Example: writing two files 7-14
7.3.3.2 Example: reading two files 7-15
7.4 INTERNAL RECORDS AND FILES 7-15
7.5 UNITS 7-16
7.5.1 Default units 7-17
7.5.2 Redirection to and from default files 7-17
7.6 I/O FORMATS 7-18
7.7 READ, WRITE, AND PRINT STATEMENTS 7-19
7.7.1 Control information list 7-20
7.7.2 I/O list 7-22
7.7.2.1 Input list items 7-22
7.7.2.2 Output list items 7-23
7.7.2.3 Implied DO list 7-23
7.7.3 Data transfer operation 7-24
7.7.3.1 Transferring data 7-25
7.7.3.2 Unformatted data transfer 7-25
7.7.3.3 Formatted data transfer 7-25
7.7.4 Output to a printer 7-26
7.7.5 Error and end-of-file conditions 7-27
7.7.6 Restrictions on input/output statements 7-27
7.6.7 I/O error recovery 7-27
7.8 OPEN STATEMENT 7-28
7.8.1 Alternatives to the OPEN statement 7-29
7.9 CLOSE STATEMENT 7-32
7.10 INQUIRE STATEMENT 7-33
7.11 DIRECT AND SEQUENTIAL FILE ACCESS 7-36
7.11.1 Direct file access 7-36

7.11 DIRECT AND SEQUENTIAL FILE ACCESS (continued)
7.11.2 Sequential file access 7-38
7.11.2.1 BACKSPACE statement 7-39
7.11.2.2 ENDFILE statement 7-39
7.11.2.3 REWIND statement 7-40
7.12 CHANGING MAXIMUM LENGTH FOR I/O LISTS AND FORMAT SPECIFICATIONS 7-40

8. INPUT/OUTPUT FORMATTING 8-1

8.1 UNFORMATTED I/O 8-1
8.2 LIST-DIRECTED I/O 8-2
8.2.1 List-directed input 8-3
8.2.2 List-directed output 8-4
8.3 FORMATTED I/O 8-5
8.3.1 FORMAT statement 8-6
8.3.2 Interaction between I/O lists and format specifications 8-7
8.3.3 Positioning by format control 8-9
8.4 FORMAT DESCRIPTORS SUMMARY 8-9
8.5 FORMATTING REAL NUMBERS (F, E, G, D) 8-13
8.5.1 Real output without exponent (F) 8-14
8.5.2 Real output with exponent (E) 8-15
8.5.3 Real output with optional exponent (G) 8-16
8.5.4 Real input (F, E, G) 8-17
8.5.5 Double-precision (D) 8-20
8.5.6 Scale factor (P) 8-20
8.6 FORMATTING OTHER DATA TYPES 8-21
8.6.1 Integer (I) 8-21
8.6.2 Complex 8-22
8.6.3 Logical (L) 8-22
8.6.4 Octal (O) (CFT77 extension) 8-24
8.6.5 Hexadecimal (Z) (CFT77 extension) 8-25
8.7 FORMATTING CHARACTER DATA (A, ', ", H) 8-26
8.7.1 Character type (A) 8-26
8.7.2 Output strings within format lines (', ") . . . 8-27
8.7.3 Hollerith character output (H) 8-27
8.8 SPECIAL-PURPOSE DESCRIPTORS (T, X, /, :, B, S, $) 8-28
8.8.1 Position control (T, TL, TR, X) 8-28
8.8.2 End of record (/) 8-29
8.8.3 Terminate format (:) 8-29
8.8.4 Interpreting blanks (BN, BZ) 8-30
8.8.5 Plus sign control (S, SP, SS) 8-30
8.8.6 Carriage control ($) (COS only, CFT77 extension) 8-31

9. CRAY I/O EXTENSIONS 9-1

9.1 BUFFER IN AND BUFFER OUT STATEMENTS (CFT77 EXTENSIONS) . 9-1
9.1.1 The UNIT function 9-3
9.1.2 The LENGTH function 9-4
9.2 RANDOM INPUT/OUTPUT OPERATIONS (CFT77 EXTENSION) 9-5
9.2.1 BUFFER IN/OUT with SETPOS 9-7
9.3 NAMELIST STATEMENT (CFT77 EXTENSION) 9-10
9.3.1 NAMELIST input 9-12
9.3.1.1 NAMELIST input variables 9-13
9.3.1.2 NAMELIST input processing 9-14
9.3.1.3 User control subroutines 9-15
9.3.2 NAMELIST output 9-16
9.3.2.1 User control subroutines 9-17

10. OPTIMIZATION 10-1

10.1 ANALYZING YOUR PROGRAM AND ITS PERFORMANCE 10-1
10.1.1 FTREF 10-1
10.1.2 Flowtrace 10-1
10.1.3 **prof** and SPY 10-2
10.1.4 Perftrace (CRAY X-MP systems only) 10-3
10.2 MULTITASKING 10-3
10.3 VECTORIZATION 10-4
10.3.1 Vectorizable loops 10-4
10.3.2 Vectorizable statements 10-4
10.3.3 Vectorizable expressions 10-5
10.3.4 Loops containing IFs 10-5
10.3.5 Recurrences 10-6

APPENDIX SECTIONS

A. CHARACTER SET A-1

B. INTRINSIC FUNCTIONS B-1

C. POWERS AND CONSTANTS C-1

D. MEMORY MANAGEMENT D-1

D.1 Changing your code: recommended method D-2
D.2 Changing your code: alternative method D-3

E. OUTMODED FEATURES E-1

E.1 HOLLERITH TYPE E-2
E.1.1 Hollerith constants E-2
E.1.2 Hollerith expressions E-4
E.1.3 Hollerith relational expresssions E-4
E.2 FORMATTED DATA ASSIGNMENT E-5
E.2.1 ENCODE statement E-6
E.2.2 DECODE statement E-6
E.3 EDIT DESCRIPTORS E-7
E.3.1 Asterisk delimiters E-7
E.3.2 Negative-valued X descriptor E-8
E.3.3 A and R descriptors for non-character types . . E-8
E.4 PUNCH STATEMENT E-11
E.5 TYPE DECLARATION E-11
E.5.1 Double declaration statements E-11
E.5.2 Type statement data length E-12
E.6 DATA STATEMENT FEATURES E-13
E.7 IF STATEMENTS E-14
E.7.1 Two-branch arithmetic IF E-14
E.7.2 Indirect logical IF E-15

F. CREATING NON-FORTRAN PROCEDURES F-1

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

G. MACHINE REPRESENTATION OF DATA G-1

G.1 INTEGER TYPE G-1
G.2 REAL TYPE G-2
G.2.1 Normalized floating-point numbers G-3
G.3 DOUBLE-PRECISION TYPE G-3
G.4 COMPLEX TYPE G-4
G.5 CHARACTER TYPE G-4
G.6 LOGICAL TYPE G-4

H. DIFFERENCES BETWEEN CFT77 AND CFT H-1

H.1 FUNCTIONAL DIFFERENCES H-1
H.2 SYNTAX AND ERROR DETECTION H-3

FIGURES

1-1 Default Files Used in Compiling and Running (UNICOS) 1-3
1-2 Default Datasets Used in Compiling and Running (COS) 1-15
2-1 Subcategories of Fortran Terms 2-9

FIGURES (continued)

2-2 Example Program Showing Fortran Structure 2-10
4-1 Array Specification and Size 4-14
4-2 Storage Sequence for Arrays in Figure 4-1 4-15
6-1 IF levels and Conditional Blocks in an IF Structure 6-3
9-1 Sample Program Using NAMELIST, with Input and Output 9-11
D-1 Memory Use under UNICOS D-2
D-2 Memory Use under COS D-2
G-1 Integer Data Formats G-1
G-2 Floating-point Data Format G-2
G-3 Double-precision Format G-3
G-4 Complex Format G-4
G-5 Character Format G-4

TABLES

1-1 Compiler Options Under UNICOS 1-10
1-2 Compiler Options Under COS 1-24
2-1 Required Order of Lines and Statements 2-6
3-1 Values Represented in Different Data Types 3-2
4-1 Possible Kinds of Arrays 4-4
4-2 Subscript Evaluation 4-11
5-1 Allowed Assignment Statements: y=x 5-4
5-2 Arithmetic Operators and Their Use in Expressions 5-5
5-3 Use of Data Types with Arithmetic Operations: +, -, *, / . . 5-10
5-4 Data Types In Exponentiation: ** 5-11
5-5 Data Types in Relational Operations: .EQ.,.NE.,.GT.,.GE.,
.LT.,LE. 5-18
5-6 Logical Operators 5-20
5-7 Meanings of Logical Operators 5-21
5-8 Allowed Logical and Masking Operations and Result Types . . . 5-24
7-1 CFT77 Input/Output Statements 7-9
7-2 Print Control Characters 7-26
7-3 OPEN Specifiers and Their Meanings 7-30
7-4 CLOSE Specifiers and Their Meanings 7-31
7-5 INQUIRE Specifiers and Their Meanings 7-34
8-1 Repeatable Format Descriptors 8-10
8-2 Nonrepeatable Format Descriptors 8-11
8-3 Format Descriptors with Data Types 8-12
8-4 format Descriptors and Data Types when SEGLDR and the EQUIV
Directive are Used 8-12
8-5 Real Output Values with F, E, and G Descriptors 8-13
8-6 Output of Exponents with E Descriptor 8-15
8-7 G*w*.*d* and G*w*.*d*E*e* and Equivalent F Descriptors 8-16
9-1 Performance Comparison of I/O Methods 9-6
9-2 Chracteristics of Random I/O Methods 9-7
A-1 Character Sets: ASCII, FORTRAN 77, CFT77 . . . . A-2
B-1 General Arithmetic Functions B-2
B-2 Type Conversion Functions B-4

TABLES (continued)

B-3 Maximum/Minimum Functions B-5
B-4 Character Functions B-5
B-5 Trigonometric Functions (Angles in radians) B-6
B-6 Exponential Functions B-7
B-7 Logarithmic Functions B-7
B-8 Boolean and Logical Functions B-8
B-9 Time and Date Functions B-10
B-10 Miscellaneous Functions B-10
B-11 Conditional Vector Merge Functions B-11
C-1 Miscellaneous Constants C-1
E-1 Data Length E-13

INDEX

NEW AND ENHANCED FEATURES

The 2.0 release of CFT77 contains many improvements to increase its usefulness. External features that you can invoke include the following:

- The INCLUDE statement, page 1-36, allows external source code files to be inserted in your program.
- The Loopmark feature, option **m**/M on pages 1-11 (UNICOS) and 1-24 (COS), marks loops in your source listing, indicating whether they have been vectorized.
- Directives BL and NOBL, page 1-32, control bottom-loading of loops, an optimization technique that can cause operand range errors.
- The I/INDEF option, pages 1-10 (UNICOS) and 1-23 (COS), prevents the use of uninitialized variables.
- The ALLOC and INTEGER directives allow local control of the control statement options -a/ALLOC and -i/INTEGER=, respectively.
- The tab character allows simpler keying of code.
- The format for cross-references, page 1-37, has been improved.
- The SHORTLOOP directive, page 1-30, although previously documented, was not effective until this release.
- The -**V** option (UNICOS only), page 1-9, generates information about the compile operation.
- The **P** and **w** options (CRAY-2 systems only), pages 1-10 and 1-12, allow more efficient use of local memory.
- The EDN keyword (COS only), page 1-21, creates a separate dataset for error listings.
- TASK COMMON storage is now available on CRAY-2 systems.

The following improvements increase the utility of the compiler or improve the compiled program's execution time:

- A new traceback format to make use of UNICOS shared code
- Improved analysis of data dependencies in loops and other improvements in vectorizing capability
- Faster compiling and improved code generation

Other Changes to This Manual

This edition of the manual includes the following changes that do not reflect changes in the compiler:

- New discussions of problems that some users have encountered in the use of CFT77:
  - The use of the **ema**/EMA option for extended memory addressing, page 1-7 (UNICOS) and 1-20 (COS); and cross-compiling, page 1-25
  - Other command-line options, compiler directives, and storage issues, page 1-3 (UNICOS) and 1-16 (COS)
  - Other storage issues: stack and static allocation, page 4-29; inputs to recursive calls, page 4-31; implicit association, 4-33
  - Vectorization of loops that use random numbers, appendix page B-13
- New discussions of system actions and utilities:
  - Debugging with the DEBUG utility, page 1-4 (UNICOS) and 1-16 (COS)
  - Source code analysis with FTREF, page 4-38
  - Program flow tracing and timing with Flowtrace, page 1-33
  - Fine-grained program timing with **prof** (UNICOS) and SPY (COS), page 10-2
  - CRAY X-MP performance monitoring with Perftrace, page 10-3
  - Specifying pure data/unblocked files for optimized I/O, page 9-8
- Improved discussions of subjects for newer users of Fortran and Cray computers:
  - Compiling and running under both Cray operating systems, pages 1-2 (UNICOS) and 1-13 (COS)
  - Tutorial on standard I/O, beginning on page 7-1
  - I/O formatting, now organized by task, section 8
  - The use of common blocks, page 4-37

THE CFT77 COMPILER 1

Fortran is a computer programming language that is well suited to mathematical problems. The CFT77 compiler compiles Fortran that conforms to the American National Standards Institute (ANSI) standard X3.9-1978, often called Fortran 77. CFT77 supports extensions to this standard to offer broader capabilities and to take advantage of the features of the CRAY-2 and CRAY X-MP computer systems; this manual identifies all CFT77 extensions to the ANSI standard. These extensions include the following:

- Most extensions supported by the CFT compiler
- Array syntax, including array sections, to allow operations on arrays without DO loops.
- Automatic arrays
- Symbolic names of up to 31 characters, including "_"

The CFT77 compiler converts Fortran language statements into machine-language instruction sequences. The compiler is supported by the Cray operating systems, UNICOS and COS.

This manual uses the following typographic conventions:

UPPERCASE	Used for Fortran statements, CFT77 compiler directives, acronyms such as "ANSI," and COS control statements.
boldface	In text and formats, used with words of mixed case or lowercase to indicate literal input and output, such as a command, file name, or table entry. In examples, used to highlight featured or changed material.
italics	Identify user-provided items and terms being defined.
[*item*]	Identifies *item* as optional. In appendix B, brackets indicate truncation of real numbers to integers.

Fortran examples are indented as in actual code: seven spaces except for directives, labels, and comment lines. Most operating system examples are indented ten spaces. Except in character strings, blanks are not significant in Fortran code and are used only to improve clarity.

1.1 USING CFT77 UNDER UNICOS

This subsection describes the use of CFT77 under the UNICOS operating system, first the basic procedure of compiling and running a program, and then the details of the **cft77** command.

1.1.1 COMPILING AND RUNNING A PROGRAM

To compile and run a typical CFT77 job under UNICOS on a Cray computer system, enter the following commands for a program contained in file **hello.f:**

1. **cft77 -e mx -V hello.f**

   The **cft77** command invokes the compiler. This must include the source file name; adding the **.f** extension or no extension to the name simplifies the naming of other files. In this example, the **-e** keyword enables these options: **m** causes listings to be written in file **hello.l** and turns on the Loopmark feature (to highlight loops in the listing); **x** causes the listing to include cross-reference information. The **-V** option causes information about the compilation to be added to the listing file and, typically, sent to your terminal.

   The compiler locates the file and compiles it, generates a binary file, and optionally generates other files (including listing file **hello.l** in this example). The binary file is used as input to the loader (step 2) to create an executable program; this file is named **hello.o** by default, where **hello** is taken from the input file name **hello.f.** Other **cft77** command options are shown in 1.1.4.

2. **segldr hello.o**

   The **segldr** command invokes the loader, which loads the program with the name of the CFT77 binary output file; the loader also loads libraries needed for your program to run. The loader's output is an executable program in file **a.out.**

3. **a.out** or **a.out > outfile < infile**

   You run your program with command **a.out.** With the first version above, your program's input and output data are contained either in files named within the program or (when you use * as a file specifier in an I/O statement) in default files **stdin** and **stdout.** With the second version above, data file **infile** is directed to **stdin,** and **stdout** is directed to **outfile,** so that * in an I/O statement effectively specifies the non-default files.

Figure 1-1 shows the process of compiling and running in simplified form, to clarify the use of the default files (assuming a source file named **hello.f**). Each default file can be renamed, replaced, or redirected.

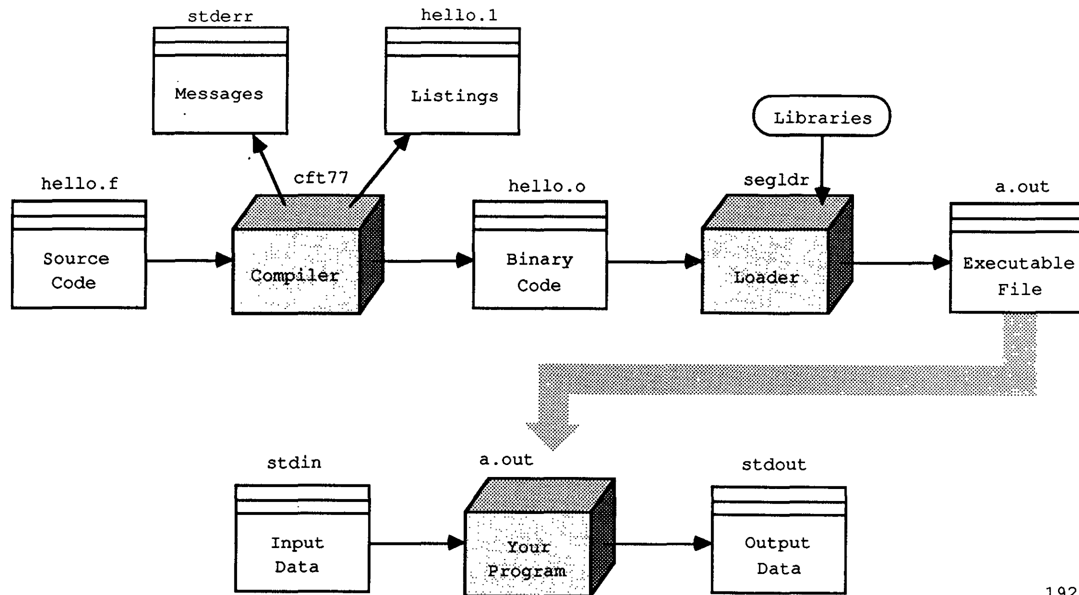


Figure 1-1. Default Files Used in Compiling and Running

1.1.2 ISSUES FOR SUCCESSFUL USE OF CFT77

This subsection describes aspects of CFT77 that differ from some users' assumptions or expectations. Also see appendix H.

The following points involve command-line options:

- Unlike CFT, CFT77 does not write a listing by default. You get a listing by including, on the **cft77** command, **-e** followed by listing options **g**, **h**, **m**, **s**, **x**, or **L** (see table 1-1). This is shown in step 1 in 1.1.1.

- The default listing file is not **stdout**. For example, the command **cft77 -e mx hello.f > listfile** writes a listing in file **hello.l** and nothing in **listfile**. To use **listfile**, include **-l listfile** in the command.

- On machines that offer extended memory addressing (EMA), you must include the **-C,ema** option to make use of the extra address space.

- A single compilation does not generate both a CAL file and a binary load file. That is, when **-s** or **-eS** appears on the command line, no binary file is generated, even if **-b** also appears. This differs from CFT.

CFT77 compiler directives are used differently than those for CFT, as follows:

- When an option is controlled by both a command line option and a pair of compiler directives (such as **-e f** and FLOW/NOFLOW), the use of the command option causes the compiler to ignore any directives related to that option. This differs from CFT.

- Each CFT77 compiler directive, other than those applying to listings, applies until the end of a program unit unless it is cancelled by a subsequent directive. (Some CFT directives apply only within an optimization block.)

Some aspects of CFT77 data storage differ from CFT and other compilers. These compilers use conventions and assumptions, not specified in the ANSI standard, that conflict with CFT77 optimization techniques. The differences are the following:

- Local data (an entity that can be referenced by only one subprogram) is not necessarily preserved between subprogram calls unless you use the SAVE statement (see 4.5.4).

- Your program must not attempt to associate data implicitly, based on assumed storage sequences other than those resulting from common block declarations and arrays. See 4.5.5 and 4.5.2.

1.1.3 PREPARING FOR DEBUGGING UNDER UNICOS

Several debugging utilities are offered for use under UNICOS, as described in the Symbolic Debugging Package Reference Manual, CRI publication SR-0112. If your job aborts, a starting point for debugging is to use the **debug** command, which provides a program summary, traceback, and variable dump.

To use any Cray debugger, including **debug**, compile your program specifying **-e D** to generate a symbol table and **-o off** to disable optimization. Then create a file, **mapdir**, containing the directive **map=full**. The following sequence would then create a debug traceback and a load map. (**Bold** type indicates what you type.)

```
cft77 -e D -o off hello.f
segldr mapdir hello.o > mapfile      /* Directive in file
a.out
error message, core dumped
debug > hello.dbg
```

The load map contained in file **mapfile** shows the starting addresses for your program's routines and other information needed for interpreting the debug traceback.

1.1.4 THE **cft77** COMMAND UNDER UNICOS

Keywords in the **cft77** command can be in any order, separated by spaces; space between a keyword and its argument is optional. If a keyword and option are omitted from the statement, the compiler uses a default value.

If an entry in the command is not a recognized keyword, the compilation is aborted. If a keyword option is unrecognized, duplicated, or in conflict with another option, the compilation is usually aborted.

If conflicting list output options appear on the **cft77** command, a warning message appears in the log file, and the option with the highest precedence is used, as follows:

1. -l **0** (highest)
2. **L** option
3. **m** option (includes the equivalent of **s**; allows **x**, **g**)
4. **h** option
5. **g**, **s**, **x** options
6. CDIR$ (lowest)

Thus, if -**es** is specified, CDIR$ NOLIST is ignored.

Format of the **cft77** command:

```
cft77 [-a alloc] [-b binfile] [-d offstrng] [-e onstrng] [-i intlen]
  [-l listfile] [-m msglev] [-o optim] [-s calfile] [-t trunc]
  [-C cpu,hdw] [-V] [--] file.f
```

The command showing all default values is as follows. *file*.**f** must be specified, and other appearances of *file* use the same name.

cft77 -a static -b *file*.**o -d ADLPSafgjoswx -e Bpqr -i46 -l** *file*.**l**
-m3 -o full,nozeroinc -t0 -- *file*.**f**

-s defaults to no file. The **-C** default comes from the operating system, except that **-C,ema** is required to make use of additional address space available with EMA hardware. **-l** *file*.**l** is applicable only if a listing option such as **s** appears after **-e**; all listing options are disabled by default.

-a *alloc*

Default: **static**

Specifies the memory allocation method for entities in memory, and supersedes the ALLOC directive. Storage allocation is discussed in subsection 4.5.2. *alloc* can be one of the following:

alloc	Description
static	All memory is statically allocated†; a stack is not used. Variables in subprograms are not always preserved between calls, unless the SAVE statement is used (see 4.5.4).
stack	All entities are allocated on a stack except constants and entities in a DATA statement, SAVE statement, or a common block, which are statically allocated.†

-b *binfile*

Default: *file*.**o**, taken from input file name *file*.**f**

Creates file *binfile* (if it does not already exist), on which the compiler writes the binary version of your program, which can be loaded and run. With **-b 0** or **-d B**, no binary load files are written. **-b** has no effect if **-s** or **-eS** appears on the command line; no binary file is generated.

-C *cpu,hdw*

Default: obtained from the operating system; but you must include **-C,ema** to use extra address space on machines with EMA

Specifies the mainframe type and optional characteristics of the hardware running the generated code (does not apply to the CRAY-2 computer system). Because your system's hardware defaults might differ from the generic defaults for a CPU type, the *cpu* value should be used only for cross-compiling; this is discussed in 1.3. To see what hardware is included on your machine, enter the **target** command with no options.

To specify a hardware option when compiling a program to run on the same machine, use the form **-C,*hdw***. Multiple *hdw* options are separated by commas. The *hdw* options are as follows:

† TASK COMMON variables and some compiler-generated temporary data, such as automatic arrays and array temporaries, are allocated on the heap.

hdw	Target Machine Does/Does Not Have
ema | **noema**	Extended memory addressing
cigs | **nocigs**	Compressed index and gather-scatter hardware
vpop | **novpop**	Vector population count functional unit

You must include **-C,ema** to make use of the extra memory space available on computers equipped with extended memory addressing (EMA). (This generates overhead.) The extra space is necessary if code plus local data exceeds 2 Mwords or if code plus local data plus Common Memory exceeds 4 Mwords. When your listing says EMA[<4MW], your machine is equipped with EMA hardware, but you have not used the **ema** option, and are therefore restricted to the 4 Mwords memory size. These options are discussed in more detail in 1.3.

CPU and hardware information can also be specified by the **target** command, which can apply to a group of compilations. (See the UNICOS User Commands Reference Manual, publication SR-2011.)

-d *offstring*

Default: **ADLPSafgjoswx**

Disables compile options; *offstring* can include any of the flags listed in table 1-1.

-e *onstring*

Default: **Bpqr**

Enables compile options; *onstring* can include any of the flags listed in table 1-1.

-i *intlen*

Default: **46**

Specifies 64-bit or 46-bit integer arithmetic; *intlen* can be either 64 or 46. 46-bit arithmetic is faster, but 64-bit arithmetic permits larger values. See 3.2. **-i** supersedes the INTEGER directive.

-l *listfile*

Default (only with one or more options or directives shown below):
file.**l**, taken from input file name *file*.**f**

Creates file *listfile* to receive list output. Listings are written only when enabled by **-e** options **L**, **g**, **h**, **m**, **s**, or **x**, which are all off by default (see table 1-1), or by the LIST directive (see 1.4.4.2). The default name *file*.**l** uses *file* from *file*.**f** on the command.

-m *msglev*

Default: **3** (only warning and error messages issued)

Indicates the lowest message level to be issued. For example, **-m2** allows Caution, Warning, and Fatal messages to appear. $0 \le msglev \le 4$. **-m0** allows all messages; fatal errors are never suppressed. Messages are sent to file **stderr** (which is sent to your terminal in many installations) and to the listing file, if enabled (see -l). The message levels are as follows:

Level	Type	Description
0	Comment	Inefficient programming
1	Note	Possible compiler problems
2	Caution	Possible user error
3	Warning	Probable user error
4	Error	Fatal error

-o *optim*

Default: **full,nozeroinc**

Specifies optimization options. *optim* can be **full, off,** or **novector;** and **zeroinc** or **nozeroinc.** The **off** option speeds up compilation considerably, at the expense of execution speed.

full | novector | off

These options cause the compiler to attempt, respectively, all optimizing and vectorizing, scalar optimizing only, and no optimizing or vectorizing. With **full** (the default), compiler directives for vectorizing are recognized. Even when **off** is specified, some processes occur that can be broadly termed "optimizing," such as scheduling and register allocation.

nozeroinc | zeroinc

nozeroinc, the default, improves execution time by assuming that constant increment variables (CIVs) are not incremented by variables with the value 0.

zeroinc adds runtime checks for zero increments of CIV increments in DO-loops.

-s *calfile*

Default: No file or option **S**

Creates file *calfile* (if it does not already exist) to receive Cray Assembly Language (CAL) output. This file can be manually modified to be input to the CAL assembler. DATA statements are not supported. No binary file is generated when **-s** or **-eS** is used, even if **-b** also appears.

-t *trunc*

Default: **0**

Indicates the number of bits to be truncated; the range is $0 \leq trunc \leq 47$. This option specifies truncation for all floating-point results but does not truncate double-precision results, function results, or constants. Truncated bits are set to 0.

-v

Default: No log file

Enables information concerning the compile operation to be sent to file **stderr** (which is sent to your terminal at many sites) and to the listing file, if enabled (see **-l**). Does not go to file **stdout**.

***file*.f**

No default; must be specified.

Specifies the file containing your Fortran code. With the **.f** suffix or with no suffix, default file names add a suffix to the name *file*. With other suffixes (such as **.pgm**), default file names add suffixes following the original suffix (for example, **hello.pgm.l**).

1.1.5 COMPILER OPTIONS UNDER UNICOS

Table 1-1 shows CFT77 compiler options, which are turned on or off with the **-e** and **-d** strings in the **cft77** command. The options establish settings throughout an executable program.

Compiler directives described in 1.4.5 can turn some of the same options on or off within programs, but only options not included in the **-e** or **-d** string; if an option appears in one of the strings, directives for that option are ignored. Some other compiler actions are not set by **cft77** command options but only by compiler directives (see 1.4).

Table 1-1. Compiler Options Under UNICOS

Option	Default	Description
A	**-d**	Generates messages to note all non-ANSI usages.
B	**-e**	Enables creation of a binary object file; that is, **-dB** disables the object file. See parameter **-b**.
D	**-d**	Generates a symbol table for the debugger on the file specified by **-b** *binfile*. Default is *file*.**o**, where *file* is specified by *file*.**f** in the command.
I	**-d**	Causes uninitialized memory to be set to an undefined value. This causes an error to occur when an uninitialized variable is used, such as in a floating-point operation or as an array subscript.
L	**-d** (output enabled by other options)	Enables all available kinds of listings, except Loopmark (option **m**). These include generated code, full cross-references with unreferenced symbols, and vectorization information. If **L** is not specified, use options **g**, **h**, **s**, or **x** to select specific kinds of output. Supersedes LIST and NOLIST directives.†
P	**-d**	CRAY-2 systems only. Generates code to cause paging of Local Memory on entry to and exit from each subprogram. Also see option w.††

† The precedence of listing options and directives is discussed at 1.1.4 introduction. Listings are written to the file specified by the -l keyword, or to *file*.l corresponding to the source file *file*.**f**.

†† Options P and w are used when programs with many subprograms cannot otherwise be loaded because they require more than the maximum available Local Memory. These options can slow execution because of extra transfers between Local and Common Memory.

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

Option	Default	Description
S	**-d**	Creates CAL file *file*.**s**, where *file* is specified by *file*.**f** in the command. Parameter **-s** creates a file with a non-default name, which overrides option **S**.
a	**-d**	Aborts job after compilation if any program unit contains a fatal error.
f	**-d**	Generates Flowtrace for the entire compilation unit. Supersedes FLOW and NOFLOW directives. See 1.4.5.1.
g	**-d**	Enables listing of generated code to the output file (*file*.**l** or the file named by **-l**). Not needed if **-eL** is used; supersedes CODE and NOCODE directives.†
h	**-d**	Enables listing of first statement in each program unit, and error messages. Superseded by **L** and **m** options; supersedes **g**, **s**, and **x**.†
j	**-d**	Causes at least one execution of each DO loop whose DO statement is executed.
m	**-d**	Enables the Loopmark option, which marks each loop in the source listing and indicates loop type, as follows: **V**, vector loop; **Vs**, short vector loop; **Vc**, conditional vector loop; **S**, scalar loop. This option generates a complete source listing without the use of the **s** option; if the **s** option is also used, a message is issued concerning conflicting options.† Supersedes the **h** option.
o	**-d**	Enables runtime checking of array bounds and conformance of arrays in array expressions except those contained in formatted WRITE statements. Out-of-bounds subscripts result in a message but no error.

† The precedence of listing options and directives is discussed at 1.1.4 introduction. Listings are written to the file specified by the **-l** keyword, or to *file*.**l** corresponding to the source file *file*.**f**.

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

Option	Default	Description
p	**-e**	Allows double precision. If **-dp** is specified, the following occurs at compile time: • Double-precision declaratives are treated as real. • Double-precision functions are changed to the corresponding single precision functions. • Double-precision constants are converted as double-precision and truncated to real. • D format descriptor is changed to E.
q	**-e**	Aborts compilation when 100 fatal error messages are counted.
r	**-e**	Rounds the results on multiply operations. This option cannot be disabled on CRAY-2 systems.
s	**-d**	Enables listing of source code to the output file (*file*.l or file named by -l). Supersedes LIST and and EJECT directives; not needed with options L or **m**; superseded by option **h**.†
x	**-d**	Enables cross reference listing to the output file (*file*.l or the file named by -l). Not needed with option L; superseded by option **h**.†
w	**-d**	CRAY-2 systems only. Causes the use of Common Memory for many purposes in which Local Memory would be used otherwise. Also see option P.††

† The precedence of listing options and directives is discussed at 1.1.4 introduction. Listings are written to the file specified by the -l keyword, or to *file*.l, corresponding to the source file *file*.f.

†† Options P and w are used when programs with many subprograms cannot otherwise be loaded because they require more than the maximum available Local Memory. These options can slow execution because of extra transfers between Local and Common Memory.

1.2 USING CFT77 UNDER COS

This subsection describes the use of COS for running your Fortran program. Consult the COS Reference Manual, publication SR-0011, concerning the subjects mentioned.

1.2.1 PREPARING YOUR PROGRAM AS A COS JOB

The CFT77 compiler translates your Fortran source code into binary object code. A single job can compile, load, and run your program, or these steps can be performed by separate jobs.

To run a job, you submit to COS a dataset containing job control language (JCL), which consists of COS control statements. This dataset can also include a file containing your source program, and one or more files of input data for your program (but the JCL file must be first). Alternatively, those files can be in other datasets, which must be made *local* (available) to your job (see 1.2.3). (If your program is contained in a separate dataset, specify it on the CFT77 command as I=*name*.)

Example:

Typically, you will prepare your input dataset on a front-end computer and submit it to COS by means of the front-end CRSUBMIT command or its equivalent. To compile and run your program in one operation, using only the default input and output datasets, a job dataset could contain the following. Notice that each statement ends with a period; /EOF marks the end of a file.

```
JOB,JN=TEST,US=U1234.
ACCOUNT,AC=5678,UPW=JOE.
CFT77,ON=MX.
SEGLDR,GO.
/EOF
   Fortran source file
/EOF
   input data file
/EOF
```

The JOB statement above is a required statement that defines the job to COS. At the minimum, it must contain a JN parameter to assign the job a name. The user name (US) parameter is required at many sites.

The ACCOUNT control statement presents your account number and often your password; these may be required by a site before access is granted.

The CFT77 statement causes the compiler to be loaded and executed, and is therefore not needed if your program has already been compiled. In this example, the ON= keyword enables these options: M causes listings to be written to dataset $OUT and turns on the Loopmark feature (to highlight loops in the listing); X causes the listing to include cross-reference information.

The SEGLDR statement with the GO parameter loads and runs your compiled program; it also accesses the CFT77 runtime library.

1.2.2 RUNNING YOUR JOB UNDER COS: OPERATIONS AND DATASETS

The file you submit to COS becomes input dataset $IN. COS executes the commands in the dataset's JCL file. When COS executes the CFT77 control statement, it invokes the CFT77 compiler, which creates a relocatable binary output dataset and an optional listing dataset.

The compiler writes the binary version of your program to a dataset, by default $BLD, and writes source code and other listings, if enabled, to another dataset, by default $OUT (see L= keyword in 1.2.6). The loader, SEGLDR or LDR, can then load the binary version, to create an executable file named $ABD by default; your program can then be executed. As shown in the example in 1.2.1, the loader can execute it automatically; or you can invoke the program by using the name of its executable file as a JCL statement. When the program runs, COS writes a log file to $OUT; the log file contains information about the job and the system.

When I/O statements in your program use * as a unit identifier, data is transferred from $IN or to $OUT. Therefore, if you compile and run a program in the same job, $OUT can contain your program's output, source listing, error messages, and log file. In many installations, $OUT is automatically transferred to your front-end computer.

To clarify the use of the default datasets, figure 1-2 shows, in simplified form, the process of compiling and running a COS job. Files for source code, output listings, input data, and output data can be replaced by datasets that you specify, as described in the following subsection.

1.2.3 USING DATASETS THAT ARE SPECIFIC TO YOUR JOB

Any nondefault datasets your job needs, such as a source file or data file not included in $IN, must be made *local* (available) to your job. To do this, you need to add control statements to the JCL shown in 1.2.1. For a file on your front-end computer, after the ACCOUNT statement insert FETCH,DN=*name*. For a dataset on the Cray computer system, insert ACCESS,DN=*name*. Other options for these statements are shown in the COS Reference Manual, publication SR-0011.

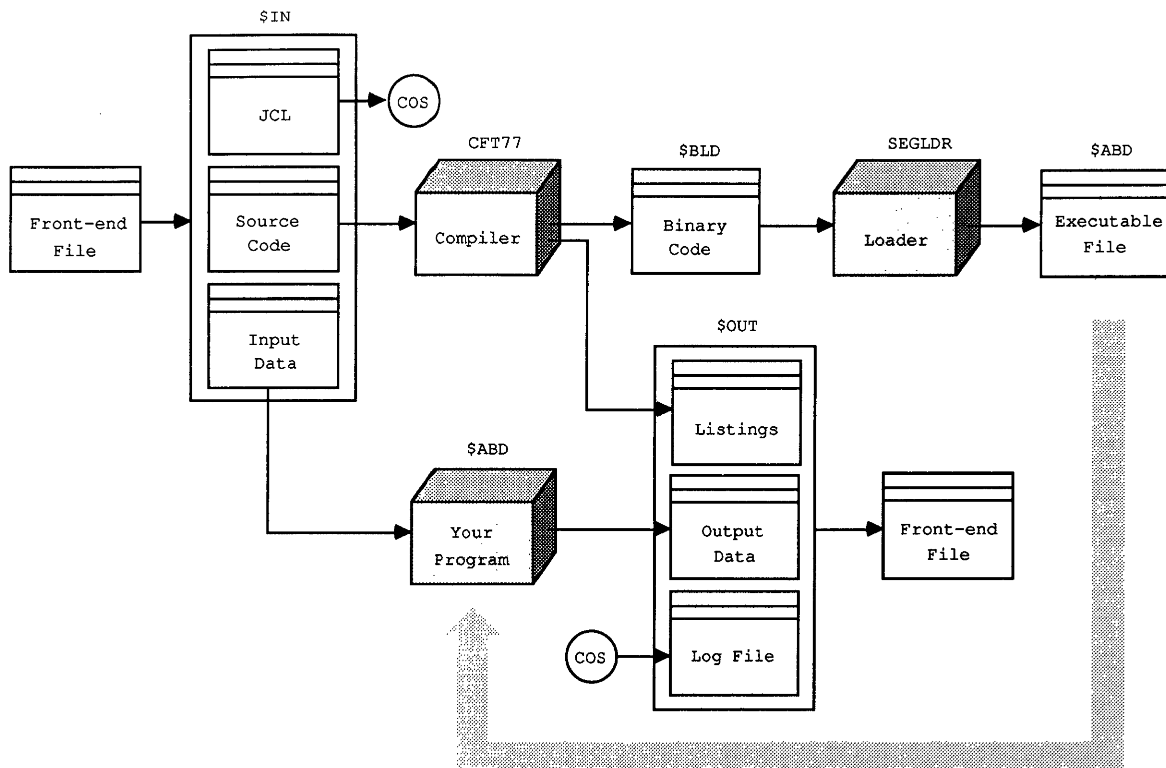


Figure 1-2. Default Datasets Used in Compiling and Running

Similarly, if you use a nondefault dataset for output, you need a control statement to specify what should be done with it; otherwise it is deleted. To transfer a dataset to your front-end computer, after the SEGLDR statement insert DISPOSE,DN=*name*. To preserve a dataset on the Cray computer system, insert SAVE,DN=*name*.

Example:

```
FETCH,DN=MYIN.
ACCESS,DN=MYPROG.
CFT77,ON=MX,I=MYPROG.
SEGLDR,GO.
DISPOSE,DN=MYPRINT.
SAVE,DN=MYOUT.
/EOF
```

The above JCL file is submitted to COS in dataset $IN. FETCH brings data file MYIN from the front end and makes it local to the job. ACCESS makes a Cray-resident dataset local to the job; the dataset is MYPROG, which contains a Fortran source program. The CFT77 statement specifies this dataset as the input to the compiler. DISPOSE sends dataset MYPRINT to the front end, and SAVE makes dataset MYOUT permanent on the Cray system.

Many Cray installations use $IN and $OUT for transfers with your front-end computer. Some Fortran programs were originally written with other assumptions about the use of the default files, in statements such as WRITE(*,50).... For such programs, you can copy between $IN or $OUT and other Cray-resident datasets; that is, you can change the JCL and leave the Fortran as it is; see 7.5.2.

1.2.4 PREPARING FOR DEBUGGING UNDER COS

Several debugging utilities are offered for use under COS, as described in the Symbolic Debugging Package Reference Manual, publication SR-0112. If your job aborts, a starting point for debugging is to use the DEBUG utility, which provides a program summary, traceback, and variable dump.

To use any Cray debugger, including DEBUG, compile your program specifying DEBUG to generate a symbol table and OPT=OFF to disable optimization. Then insert statements following the SEGLDR statement, so that the JCL file reads as follows:

```
...
CFT77,OPT=OFF,DEBUG.
SEGLDR,CMD='MAP=FULL',GO.
EXIT.
DUMPJOB.
DEBUG.
/EOF
```

The EXIT statement above specifies a course of action if your program is not successfully loaded or executed. DUMPJOB causes your aborted job to be dumped to dataset $DUMP, and DEBUG causes debugging information to be written in dataset $OUT. Information in the debug traceback is interpreted with the aid of the load map (also in $OUT, generated by the directive on the SEGLDR control statement).

1.2.5 ISSUES FOR SUCCESSFUL USE OF CFT77

This subsection describes aspects of CFT77 that differ from some users' assumptions or expectations. Also see appendix H.

The following points involve command-line options:

- Unlike CFT, CFT77 does not write a listing file by default. As shown in the CFT77 control statement in 1.2.1, you get a listing by including ON= followed by listing options G, H, M, S, or X (see table 1-2), or the LIST keyword.

- On machines that offer extended memory addressing (EMA), you must include the CPU=:EMA option to make use of the extra address space.

* A single compilation does not generate both a CAL file and a binary load file. That is, when C=*cdn* appears on the command line, no binary file is generated, even if B=*bdn* also appears. This differs from CFT.

CFT77 compiler directives are used differently than those for CFT, as follows:

* When an option is controlled by both a control statement option and a pair of compiler directives (such as ON=F and FLOW/NOFLOW), the use of the control statement option causes the compiler to ignore any directives related to that option. This differs from CFT.

* Each CFT77 compiler directive, other than those applying to listings, applies until the end of a program unit unless it is cancelled by a subsequent directive. (Some CFT directives apply only within an optimization block.)

Some aspects of CFT77 data storage differ from CFT and other compilers. These compilers use conventions and assumptions, not specified in the ANSI standard, that conflict with CFT77 optimization techniques. The differences are the following:

* Local data (an entity that can be referenced by only one subprogram) is not necessarily preserved between subprogram calls unless you use the SAVE statement (see 4.5.4).

* Your program must not attempt to associate data implicitly, based on assumed storage sequences other than those resulting from common block declarations and arrays. See 4.5.5 and 4.5.2.

1.2.5 THE CFT77 CONTROL STATEMENT UNDER COS

The CFT77 compiler is loaded and executed when a CFT77 control statement is entered interactively or encountered in the JCL file in the $IN dataset.

Keywords can be in any order. If a keyword and option are omitted from the statement, the compiler uses a default value. If an entry on the control statement is not a recognized keyword, the job is aborted. If a keyword option is unrecognized, duplicated, or in conflict with another option, the job is usually aborted.

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 alphanumeric characters.

If conflicting list output options appear on the CFT77 control statement, a warning message appears in the log file, and the option is used with the highest precedence as follows:

1. L=0 (highest)
2. LIST option
3. M option (includes provision equivalent to S; allows C, X, G)
4. H option
5. G, S, X options
6. CDIR$ (lowest)

Thus, if ON=S is specified, CDIR$ NOLIST is ignored.

Format of the CFT77 control statement:

```
CFT77, ALLOC=a, B=binarydn, C=caldn, CPU=cpu:hdw, E=msglev,
  EDN=errordn, I=inputdn, INTEGER=n, L=listingdn, OFF=string,
  ON=string, OPT=optim, TRUNC=n, DEBUG, INDEF, LIST, STANDARD.
```

The control statement showing all default values is as follows:

```
CFT77, ALLOC=STATIC, B=$BLD, E=3, EDN=$OUT, I=$IN, INTEGER=46, L=$OUT,
OFF=AFGHJOSX, ON=PQR, OPT=FULL:NOZEROINC, TRUNC=0.
```

C defaults to no file. The CPU default comes from the operating system, except that CPU=:EMA is always required to make use of additional address space available with EMA hardware. L=$OUT applies only if a listing option such as S appears after ON=; all listing options are disabled by default (see table 1-2).

ALLOC=*a*

Default: STATIC

Specifies the memory allocation method for entities in memory, and supersedes the ALLOC directive. Storage allocation is discussed in subsection 4.5.2. *alloc* can be one of the following:

alloc	Description
STATIC	All memory is statically allocated†; a stack is not used. Variables in subprograms are not always preserved between calls, unless the SAVE statement is used (see 4.5.4).
STACK	All entities are allocated on a stack except constants and entities in a DATA statement, SAVE statement, or a common block, which are statically allocated.†

B=*binarydn*

Default: $BLD

Creates dataset *binarydn* (if it does not already exist), on which the compiler writes binary load modules. If B=0, no binary load files are written. The B= keyword has no effect if C=*caldn* also appears; no binary file is generated.

C=*caldn*

Default: No file

Creates dataset *caldn* (if it does not already exist) to receive Cray Assembly Language (CAL) output. This file can be manually modified to be input to the CAL assembler. DATA statements are not supported. No binary file is generated with C=*cdn*, even if B=*bdn* also appears.

CPU=*cpu*:*hdw*

Default: Obtained from the operating system; but you must include CPU=:EMA to use extra address space on machines with EMA

Specifies the mainframe type and optional characteristics of the hardware running the generated code. Because your system's hardware defaults might differ from the generic defaults for a CPU type, the *cpu* value should be used only for cross-compiling; this is discussed in subsection 1.3. To see what hardware is included on your machine, use the TARGET control statement.

† TASK COMMON variables and some compiler-generated temporary data, such as automatic arrays and array temporaries, are allocated on the heap.

To specify a hardware option when compiling a program to run on the same machine, use the form CPU=:*hdw*. Multiple *hdw* options are separated by colons. The *hdw* options are as follows:

hdw	Target Machine Does/Does Not Have
EMA, NOEMA	Extended memory addressing
CIGS, NOCIGS	Compressed index and gather-scatter hardware
VPOP, NOVPOP	Vector population count functional unit

You must include CPU=:EMA to make use of the extra memory space available on computers equipped with extended memory addressing (EMA). (This generates overhead.) The extra space is necessary if code plus local data exceeds 2 Mwords or if code plus local data plus Common Memory exceeds 4 Mwords. When your listing says EMA[<4MW], your machine is equipped with EMA hardware, but you have not used the :EMA option, and are therefore restricted to the 4 Mwords memory size. These options are discussed in more detail in 1.3.

CPU and hardware information can also be specified by the TARGET command in a JCL file (see the COS Reference Manual, publication SR-0011). The TARGET command allows specifying CPU information once for a group of compilations.

E=*msglev*

Default: 3 (only WARNING and ERROR messages issued)

Indicates the lowest message level to be issued. For example, E=2 allows CAUTION, WARNING, and FATAL messages to appear. 0 ≤ *msglev* ≤ 4. E=0 allows all messages, and fatal errors are never suppressed. The message levels are as follows:

Level	Type	Description
0	Comment	Inefficient programming
1	Note	Possible problems with other compilers
2	Caution	Possible user error. (Example: no path to a statement)
3	Warning	Probable user error. (Example: an array with too few subscripts)
4	Error	Fatal error

EDN=*errordn*
Default: $OUT

Creates dataset *errordn* (if it does not already exist) to receive error message listings. Messages written to dataset *listingdn* are not affected.

I=*inputdn*
Default: $IN

Specifies the name of the dataset containing Fortran source code to be input to the compiler. The default, $IN, refers to the same dataset that contains the JCL. If the source code is in a different dataset, that dataset must be local; it can be established by a JCL command such as ACCESS (for a Cray-resident dataset) or FETCH or ACQUIRE (for a front-end file).

INTEGER=*n*
Default: 46

n can equal 64 or 46, to specify 64-bit or 46-bit integer arithmetic. 46-bit arithmetic is faster, but 64-bit arithmetic permits larger values. See 3.2. INTEGER supersedes the INTEGER directive.

L=*listingdn*
Default: $OUT (only with options or directives shown below)

Creates dataset *listingdn* (if it does not already exist) to receive listed output. Unlike $OUT, this dataset must be explicitly provided for in the JCL file, such as with a SAVE or DISPOSE command.

Listings are written only when enabled by ON options C, G, H, M, S, or X, which are all off by default (see table 1-2), the LIST keyword, or the LIST directive (see 1.4.4.2). L=0 disables all listing options and directives.

OFF=*string*
Default: AFGHMJOSX

Disables compile options (see table 1-2); *string* can include any of the flags listed in table 1-2.

ON=*string*
Default: PQR

Enables compile options (see table 1-2); *string* can include up to 12 characters representing options to be enabled.

OPT=*optim*
Default: FULL:NOZEROINC

Specifies optimization options. *optim* can be FULL, OFF, or NOVECTOR; and ZEROINC or NOZEROINC. The OFF option speeds up compilation considerably, at the expense of execution speed.

FULL, NOVECTOR, OFF

These options cause the compiler to attempt, respectively, all optimizing and vectorizing, scalar optimizing only, and no optimizing or vectorizing. With FULL (the default), compiler directives for vectorizing are recognized. Even when OFF is specified, some processes occur that can be broadly termed "optimizing," such as scheduling and register allocation.

NOZEROINC, ZEROINC

NOZEROINC, the default, improves execution time by assuming that constant increment variables (CIVs) are not incremented by variables with the value 0.

ZEROINC adds runtime checks for zero increments of CIV increments in DO-loops.

TRUNC=*n*
Default: 0

Indicates the number of bits to be truncated; the range is $0 \le trunc \le 47$. This option specifies truncation for all floating-point results but does not truncate double-precision results, function results, or constants. Truncated bits are set to 0.

DEBUG
Default: No table

Generates symbol table for debugger on the dataset specified by B=*binarydn*.

INDEF

Default: Action is not performed.

Causes allocated but uninitialized memory to be set to an undefined value. This causes an error to occur when an uninitialized variable is used, such as in a floating-point operation or as an array subscript.

LIST

Default: No listing or listings selected by other options

Enables all available kinds of output listing except Loopmark (specified by ON=M). These include generated code, full cross-references with unreferenced symbols, and vectorization information. If LIST is not specified, listings are specified by ON= options G, H, S, or X (see table 1-2), or by the LIST directive (see 1.4.4.2). The LIST keyword supersedes these; their precedence is discussed in 1.2.1 introduction. Also see the L= keyword.

STANDARD

Default: Extensions are not noted.

Notes extensions to Fortran 77 syntax.

1.2.7 COMPILER OPTIONS UNDER COS

Table 1-2 shows CFT77 compiler options, which are turned on or off with the ON and OFF strings in the CFT77 control statement. The options establish settings throughout an executable program. Compiler directives described in 1.4.5 can turn many of the same options on or off within programs, but only those options not included in the ON or OFF string; if an option appears in one of the strings, directives for that option are ignored. Some other compiler actions are not set by control statement options but only by compiler directives (see 1.4).

Table 1-2. Compiler Options Under COS

Option	Default	Description
A	OFF	Aborts job after compilation if any of the program units contains a fatal error
F	OFF	Generates Flowtrace for the entire compilation unit. Supersedes FLOW and NOFLOW directives. See 1.4.3.
G	OFF	Enables listing of generated code to the output dataset (*listingdn* or $OUT). Supersedes CODE and NOCODE directives; not needed if LIST is specified.†
H	OFF	Enables listing of first statement in each program unit, and error messages. Superseded by LIST and M; supersedes G, S, and X.
J	OFF	Causes execution of all DO loops whose DO statements are executed
M	OFF	Enables loopmark option, which marks each loop in the source listing and indicates loop type, as follows: V, vector loop; Vs, short vector loop; Vc, conditional vector loop; S, scalar loop. This option generates a complete source listing without the use of the S option; if the S option is also used, a message is issued concerning conflicting options. Supersedes the H option. †
O	OFF	Enables runtime checking of array bounds and conformance of arrays in array expressions except those contained in formatted WRITE statements. Out-of-bounds subscripts result in a message but no error.
P	ON	Allows double precision. If OFF=P is specified, the following occurs at compile time: • All double-precision declaratives are treated as real. • Double-precision functions are changed to the corresponding single precision functions. • Double-precision constants are converted as double-precision and truncated to real. • D format descriptor is changed to E.

† The precedence of listing options and directives is discussed at 1.2.6 introduction. Listings are written to the dataset specified by the L=*listingdn* keyword, or by default to dataset $OUT.

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

Option	Default	Description
Q	ON	Aborts compilation when 100 fatal error messages are counted
R	ON	Rounds the results on multiply operations
S	OFF	Enables a listing of source code to the output dataset (*listingdn* or $OUT). Supersedes LIST, NOLIST, and EJECT directives; not needed if LIST or M is specified.†
X	OFF	Enables cross reference listing to the output dataset ($OUT or *listingdn*). Not needed if LIST is specified.†

† The precedence of listing options and directives is discussed at 1.2.6 introduction. Listings are written to the dataset specified by the L=*listingdn* keyword, or by default to dataset $OUT.

1.3 CROSS-COMPILING USING THE -C OR CPU= KEYWORD

This subsection describes the use of the *cpu* options and *hdw* (particularly **ema**/EMA) options, which are specified with the -C or CPU= keyword. The issues described are of interest primarily for compiling a program on one machine to run on another machine.

When you specify a *cpu* option, the defaults for *hdw* options are the minimum configuration for that CPU type. Therefore, if you are compiling on a CRAY X-MP system that has gather-scatter hardware on by default, the specification **-C cray-xmp** or CPU=CRAY-XMP has the effect of turning off gather-scatter, because this is the default for this CPU type.

Except for vector population count, all *hdw* defaults are "off" for all *cpu* options. The following list of *cpu* options is divided according to the default for the **vpop** option.

CPU types with a **novpop**/NOVPOP default:

-C *cpu*	C=*cpu*	Generates Code For
cray-1m	CRAY-1M	CRAY-1 M computer systems
cray-1	CRAY-1	CRAY-1 computer systems
cray-1a	CRAY-1A	
cray-1b	CRAY-1B	

CPU types with a **vpop**/VPOP default:

-C *cpu*	C=*cpu*	Generates Code For
cray-1s	CRAY-1S	CRAY-1 S computer system
cray-xmp	CRAY-XMP	CRAY X-MP computer systems; code generated with this option runs on one-, two-, or four-processor CRAY X-MP computer systems.
cray-x1	CRAY-X1	One-processor CRAY X-MP computer systems
cray-x2	CRAY-X2	Two-processor CRAY X-MP computer systems
cray-x4	CRAY-X4	Four-processor CRAY X-MP computer systems

The presence of hardware for extended memory addressing (EMA) involves two separate issues: load instructions and a special addressing sequence. These are determined as follows:

- If EMA hardware is present on a machine, code running on the machine must use 24-bit load instructions, even if the extra address space is not used. The compiler therefore generates these instructions automatically if the hardware is present.

- To make use of the extra address space, your binary code must include a special addressing sequence, which adds overhead; this is generated only if you specify **-C,ema** or CPU=:EMA. The extra space is necessary if code plus local data exceeds 2 Mwords or if code plus local data plus Common Memory exceeds 4 Mwords.

For cross-compiling between an EMA machine and a non-EMA machine, both of the preceding considerations (load instructions and addressing sequence) are determined only by the command-line specification. That is, the compiler's hardware detection is overridden either by the **ema**/EMA option specified on a non-EMA machine or by **noema**/NOEMA specified on an EMA machine. Code generated on a non-EMA host for an EMA target always includes the special addressing sequence; that is, it cannot use the EMA[<4MW] default option available on EMA machines.

The listing designations EMA, NOEMA, and EMA[<4MW] are interpreted as follows:

- EMA indicates that the generated code uses both 24-bit load instructions and the special addressing sequence. This results from **-C,ema** on the **cft77** command or CPU=:EMA on the CFT77 control statement, whether or not the host machine has EMA hardware.

- EMA[<4MW] indicates the use of 24-bit loads but not the special addressing sequence. This occurs only when the host machine has EMA hardware but no specification appears on the command line.

* NOEMA indicates that the generated code includes neither 24-bit loads nor the addressing sequence, and results from either absence of EMA hardware or (when compiled on an EMA machine) from the **-C,noema** or CPU=:NOEMA specification.

1.4 COMPILER DIRECTIVES

Compiler directives are lines within source code that specify options to be performed by the compiler; they are not Fortran code. Except for EJECT, LIST, and NOLIST, directives can appear only within a program unit and apply only to that program unit. Compiler directive lines are listed in the source statement listing.

Compiler directives can turn options on or off within programs, but only those options not enabled or disabled on the **cft77** command or the CFT77 control statement (see 1.1.4 and 1.2.6). Command-line options apply to an entire program. Some options set by directives cannot be set in the compiler command.

A *compiler directive line* contains the characters CDIR$ in columns 1 through 5. Column 6 must be blank. Columns 7 through 72 contain zero or more compiler directives separated by commas. If the directive includes a list, no other directive can appear on the same line. Spaces can precede, follow, or be embedded within a compiler directive. Columns 73 through 96 are ignored. Compiler directive lines cannot have continuation lines.

The C in column 1 causes other compilers to treat CFT77 directives as comments; this helps keep CFT77 programs transportable.

CFT77 provides the following categories of compiler directives.

* Vectorization control
* Controlling scalar optimization
* Listable output control
* Localized use of options available on the command line (storage allocation, integer length, bounds checking, Flowtrace)
* Dynamic common block

If no format block is shown, directives described in this section consist of the directive name only, with no parameters.

1.4.1 VECTORIZATION DIRECTIVES

Vectorization is described in section 10. The following directives are used to control it:

- VECTOR
- NOVECTOR
- IVDEP
- VFUNCTION

1.4.1.1 Suppressing vectorization (VECTOR and NOVECTOR)

The NOVECTOR directive suppresses the compiler's attempts to vectorize loops. NOVECTOR takes effect at the beginning of the next loop and applies to the rest of the program unit unless it is superseded by a VECTOR directive.

The VECTOR directive causes the compiler to resume its attempts to vectorize loops if such attempts were suppressed by a previous vectorization directive. After a VECTOR directive is specified, DO-loops with a known trip count of one are executed in scalar mode; vectorization is attempted for those with a trip count of 4 or more or with an unknown trip count.

The VECTOR directive takes effect immediately; however, a loop is not vectorized if any statement in the loop is in the range of a NOVECTOR directive. VECTOR applies to the rest of the program unit unless it is superseded by another vectorization directive.

The scope of the VECTOR and NOVECTOR directives is a single program unit. VECTOR and NOVECTOR are superseded by the **-o**/OPT keyword. Both VECTOR and NOVECTOR directives can be specified in a single program unit.

1.4.1.2 Ignore dependencies (IVDEP)

The IVDEP directive, appearing immediately before a loop, causes the compiler to ignore vector dependencies, including explicit dependencies, in any attempts to vectorize the loop. IVDEP applies only to DO loops and affects only the loop it directly precedes. Whether or not IVDEP is used, conditions other than vector dependencies can inhibit vectorization. IVDEP is superseded by the **-o**/OPT keyword.

1.4.1.3 Vectorizable functions (VFUNCTION)

The VFUNCTION directive declares that a vector version of an external function exists. The VFUNCTION directive must precede any statement function definitions or executable statements in a program.

```
VFUNCTION f[,f]...
```

f Symbolic name of a vector external function; cannot have more than 6 characters. (This is because the % character is added at the beginning and end of the name as part of the calling sequence.)

The following rules and recommendations apply to any function *f* named as an argument in a VFUNCTION directive:

- *f* must be written in CAL and must use the call-by-register sequence.
- Arguments to *f* must be either vectorizable expressions or scalar expressions; array expressions are not allowed.
- A call to *f* can pass a maximum of seven single-word items or three double-word items. These can be mixed in any order with a maximum of seven words total.
- *f* should receive inputs from its argument list rather than from a common block.
- *f* should not change the value of its arguments or variables in common blocks. Any changed value should be for variables that are distinct from the arguments.
- *f* should not reference variables in common blocks that are also used by a program unit in the calling chain.
- *f* must not have side effects.

If the argument list for *f* 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 function reference is not in a vector loop, *f* is called with all arguments passed in S registers.

1.4.1.4 Loops with low trip counts (SHORTLOOP)

The SHORTLOOP directive, placed before a DO statement, causes the DO loop to be executed at least once and at most 64 times, allowing CFT77 to generate special code. SHORTLOOP can shorten execution time because it eliminates the runtime tests that determine whether a vectorized DO loop has been completed. If the DO loop's trip count is outside the range of 1 to 64, results are unpredictable.

1.4.1.5 Register storage across subprograms (NO SIDE EFFECTS)

The NO SIDE EFFECTS directive allows the compiler to keep information in registers across subprogram invocations without reloading the information from memory after returning from the subprogram. The directive is not needed for intrinsic functions and VFUNCTIONS.

NO SIDE EFFECTS declares that a called subprogram does not redefine any variables that are local to the calling program, passed as arguments to the subprogram, or declared in a common block.

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

f Symbolic name of a 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. It 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 are unpredictable.

The NO SIDE EFFECTS directive must precede arithmetic statement functions and executable statements in a program.

CFT77 may move invocations of a NO SIDE EFFECTS subprogram from the body of a DO loop to the loop preample 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.

1.4.2 SCALAR OPTIMIZATION DIRECTIVES

Two directives (SUPPRESS and NOBL) are available to deactivate optimization to counteract occasional side-effects of register storage and bottom-loading of loop operands.

1.4.2.1 Momentary suppression (SUPPRESS)

The SUPPRESS directive suppresses scalar optimization at the point where the directive appears (and prevents vectorization of any loop that includes SUPPRESS). At CDIR$ SUPPRESS, variables in registers are stored to memory (to be read out at their next reference), and expressions are recomputed at their next reference after CDIR$ SUPPRESS. The effect on optimization is equivalent to that of a subroutine call whose argument list includes every variable in the calling program unit. Example:

```
      CALL DUMMY (all variables)  ! Sbrtn only returns to calling unit
```

The above statement has the same effect on optimization as SUPPRESS, if subroutine DUMMY does nothing but return to the calling program unit. Optimization guarantees that all variables are stored to memory before the call to DUMMY, because they are in the argument list; and it guarantees that after the call, at the next reference, each variable must be read from memory and each expression must be recomputed.

Unlike other compiler directives, SUPPRESS takes effect only if it is on an execution path. That is, optimization proceeds normally if the directive's path is not executed because of a GOTO or IF. Example:

```
      SUBROUTINE SUB (L)
      LOGICAL L
      A = 1.0              ! A is local
      IF (L) THEN          ! SUPPRESS has no effect if L is false
CDIR$   SUPPRESS
        CALL ROUTINE()
      ELSE
        PRINT *, A
      ENDIF
      END
```

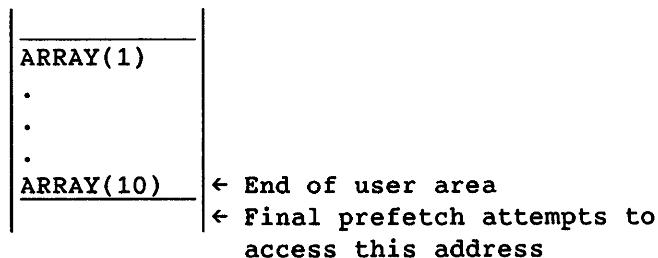
In the PRINT statement above, optimization replaces the reference to A with the constant 1, even though CDIR$ SUPPRESS appears between A=1 and the PRINT statement. The IF statement causes execution to bypass CDIR$ SUPPRESS. If SUPPRESS appears before the IF statement, A in PRINT *, A is not replaced by the constant 1.

1.4.2.2 Bottom loading of operands (BL/NOBL)

The NOBL directive disables bottom loading of loops; the BL directive causes the compiler to resume bottom loading. Bottom loading, used only on eligible scalar loops, consists of prefetching operands during each iteration for use in the next iteration.

A prefetch is performed even during the final loop iteration, because the final jump test has not been performed. If the final iteration accesses the first or last address of an array, the final prefetch attempts to access an address outside the array. If the address is outside the user program area, an operand range error can occur. To prevent this problem, use the NOBL directive to turn off bottom-loading. Example:

```
      REAL ARRAY(10)
      DO 10 I=1,10
        ...
        Y = X*ARRAY(I) + ...
        ...          ! Operand ARRAY(I+1) is fetched for next iteration
 10   CONTINUE
```



The scope of the BL and NOBL directives is a single program unit. Either directive applies for the remainder of the program unit or until the appearance of the other directive in the same program unit. Both directives can be specified in a single program unit.

1.4.3 OUTPUT LISTING DIRECTIVES

Listable output is described in 1.6. Directives have a lower precedence than the listing keywords and options in the **cft77** command and CFT77 control statement (precedence shown in 1.1.4 and 1.2.6 introductions). Directives include the following.

- EJECT inserts a page break.
- LIST and NOLIST control listing of source code.
- CODE and NOCODE control listing of binary object code.

1.4.3.1 Inserting a page break (EJECT)

A compiler directive line containing an EJECT directive is printed as the last line of the current page of source statement listing. The next page has a page header and source listings are continued. The EJECT directive has no effect if production of the source statement listing has been suppressed. EJECT is superseded by the s/S and,L/LIST options.

1.4.3.2 Listing of source program (LIST and NOLIST)

The LIST directive causes the production of a source statement listing. The NOLIST directive suppresses the production of a source statement listing.

1.4.3.3 Listing of generated code (CODE and NOCODE)

The CODE directive causes CFT77 to generate a code listing. Code is listed for the program unit in which the CODE directive occurs, and for subsequent program units until a NOCODE directive is encountered.

The NOCODE directive prevents CFT77 from producing a CFT77-generated code listing. The NOCODE directive takes effect for the entire program unit in which it is encountered, and no generated code is produced for subsequent program units until a CODE directive is encountered.

CODE and NOCODE are superseded by command options (see 1.1.4 and 1.2.6 introductions).

1.4.4 LOCALIZED CONTROL OF COMMAND OPTIONS

This subsection describes features that can be turned on and off within a program or can be specified on the command line to apply to an entire compilation.

1.4.4.1 Flowtrace (FLOW and NOFLOW)

The FLOW and NOFLOW directives control the Flowtrace feature, which prints calling and timing information about each called procedure in a program, as monitored during execution. (For a static calling tree based only on source code, use FTREF, described in 4.6.1.). Flowtrace is described in the UNICOS Performance Utilities Reference Manual, publication SR-2040, and the COS Performance Utilities Reference Manual, publication SR-0146. This subsection is a summary.

To use Flowtrace, include **-e f** on the **cft77** command or ON=F on the CFT77 control statement; or insert CDIR$ FLOW anywhere within a program unit. If you use FLOW, you can disable Flowtrace with NOFLOW; these directives let you localize the use of Flowtrace. The **f**/F option supersedes these directives. With any of these methods, you must begin your main program with a PROGRAM statement, and replace CALL EXIT or CALL ABORT with STOP or END.

On CRAY-2 computer systems, to obtain Flowtrace output you must enter a separate command after your program runs. Use **flow** for 80-column output and **flodump** for 132-column output.

Flowtrace output is written to file **stdout** under UNICOS or dataset $OUT under COS. UNICOS examples:

CRAY X-MP systems:

```
cft77 -e f hello.f
segldr hello.o
a.out > hello.flo
```

CRAY-2 systems:

```
cft77 -e f hello.f
segldr hello.o
a.out
flow > hello.flo
```

The SETPLIMQ subroutine prints a line of timing data for each CALL and RETURN statement. Insert CALL SETPLIMQ(*count*) in your program, where *count* is twice the number of CALL statements to be traced. Use directives FLOW and NOFLOW to include only areas of special interest.

The FLOWMARK subroutine, only on CRAY-2 computer systems, allows Flowtrace to treat any block of code in your program as a separate called procedure. Precede the code block with CALL FLOWMARK('*name*'L), where *name* is a string of no more than 7 characters. Follow the block with CALL FLOWMARK(0).

1.4.4.2 Array bounds checking (BOUNDS and NOBOUNDS)

The BOUNDS directive checks most array references for out-of-bounds subscripts. The NOBOUNDS directive turns off array checking. Either directive can specify particular arrays or can apply to all arrays.

The BOUNDS and NOBOUNDS directives are superseded by **-e o** in the **cft77** command or ON=O in the CFT77 control statement. The **o**/O option is global to all program units in the compilation; the BOUNDS and NOBOUNDS directives are local to the program unit in which they appear.

Bounds checking typically increases program run time and may inhibit vectorization of any DO-loop that references a checked array. If an array's last dimension declarator is *, checking is not performed on the last dimension. Arrays in formatted WRITE statements are not checked.

```
BOUNDS [arnm][,arnm]...

NOBOUNDS [arnm][,arnm]...
```

arnm Name of an array. When no array name is specified, the directive applies to all arrays.

BOUNDS remains in effect until a NOBOUNDS directive or the end of the program unit. Bounds checking can be enabled and disabled many times in a specific program unit.

1.4.4.3 Storage allocation (ALLOC)

The ALLOC directive specifies the allocation scheme for local data (defined in 4.5.2). The directive can appear anywhere in a program unit and applies only to that program unit, unlike the **-a** and ALLOC command options, which apply to the entire program and supersede the ALLOC directive. ALLOC has the following format; *a* can be either STATIC or STACK, as explained under **-a** in 1.1.4 and under ALLOC in 1.2.6.

```
ALLOC=a
```

1.4.4.4 Integer length (INTEGER)

The INTEGER directive specifies the integer length to be used within one program unit, and does not apply to subsequent program units. The directive must immediately follow the program unit's header statement (PROGRAM, FUNCTION, or SUBROUTINE). The **-i** and INTEGER command options apply to the entire program and supersede the INTEGER directive. INTEGER applies at both compile time and run time and has the following format; *n* can be either 46 or 64.

```
INTEGER=n
```

1.4.5 DYNAMIC COMMON BLOCK DIRECTIVE (DYNAMIC)

The DYNAMIC directive declares dynamic common blocks for users with dynamic common block capability (not supported by Cray loaders or operating systems other than CTSS).

```
DYNAMIC b[,b]...
```

b Name of a previously encountered common block

1.5 INCLUDE STATEMENT - INSERTING EXTERNAL SOURCE FILES

The INCLUDE statement names a file containing ASCII source code; during compilation, this file is inserted where the INCLUDE appears. INCLUDE is useful for managing code that is needed in several program units, such as common block declarations. It also lets you structure your code without the runtime overhead of calling a subprogram.

Notice that INCLUDE is a statement, not a compiler directive; like other Fortran statements, it begins on column 7.

```
INCLUDE 'file'
```

file Name of the file to be inserted. Under UNICOS, any single file name can contain up to 14 characters, and an entire path name can contain up to 128 characters. Under COS, the name can consist of up to 7 characters (ignoring blanks), and the named dataset must be local to your job (see 1.2.3).

The INCLUDE statement can appear anywhere in a program unit but cannot appear within another statement (such as logical IF) and cannot be labeled or jumped to. The line following an INCLUDE statement cannot be a continuation line. Like a compiler directive, INCLUDE is used unconditionally at compile time. INCLUDE statements can appear in the file to be inserted, up to 20 nesting levels.

The file to be inserted must not be empty, must not begin with a continuation line, and must not be any file listed on the **cft77** command or CFT77 control statement. The text in the file can cross program units; that is, it can contain the end of one program unit and the beginning of another.

Your source listing will show a notation at the beginning and end of the area of code that is inserted as the result of an INCLUDE. The notation shows the INCLUDE statement, the name of the file inserted, the name of the file containing the INCLUDE statement, and the line number of the INCLUDE statement both within the file and within the entire program.

Under COS, the external file is rewound before it is used. If the external file is $IN, it is rewound and the file containing the JCL is skipped.

1.6 LISTABLE OUTPUT

CFT77 produces a listing only when it is enabled by keywords, options, and directives shown under -l in 1.1.4 and L= in 1.2.6. The output can include the following:

Page header lines — Each page of listable output begins with a header line containing the following:

- The name of the program unit
- Current page number within the program unit
- CFT77 revision level and assembly date
- Truncation count if nonzero (see the -t and TRUNC options)
- Lists of compiler options that were on and off for this compilation (see tables 1-1 and 1-2, and the **-C** and CPU options)
- Date and time when the compilation began
- Global page number
- Status of optimization for the compiler

Source statement listings — A listing of CFT77 source statements is generated by the **s/S**, **m/M**, or **L/LIST** options. The listing is a record of all statements in the program as they are read from the source input file. Any error encountered during compilation of a statement is flagged by a line following that statement.

Two sequence numbers identify each statement's position in the program, as follows:

- The leftmost statement number reflects the sequence within the source file; at the start of an INCLUDE file, this sequence is reset to 1, and following the INCLUDE file the numbering resumes from the number preceding the INCLUDE statement.

- The second number reflects the sequence within the program unit. Statements inserted by an INCLUDE statement are numbered as if they were part of the main file; that is, the number sequence continues across the beginning and end of the INCLUDE file.

Cross reference listings	Separate tables showing address symbols, statement labels, and parameters (see 1.7). These tables are generated by the **x**/X or L/LIST option.
Messages	Up to five levels of messages are produced by CFT77, depending on the **-m** parameter on the **cft77** command or the E= parameter on the CFT77 control statement.

1.7 CROSS-REFERENCE LISTINGS

The cross-reference listings are generated by the X option or LIST keyword in the CFT77 control statement. They consist of the Symbol Cross-reference Table, the Parameter Table, and the Label Cross-reference Table.

The tables represent data types as follows:

Data Type	Representation
Integer of 64 bits	**Int64**
Integer of 46 bits	**Int46**
Real	**Real**
Double precision	**Double**
Complex	**Complex**
Character	**Char**
Logical	**Logical**
Pointer	**Pointer**

1.7.1 SYMBOL CROSS-REFERENCE TABLE

The Symbol Cross-reference Table alphabetically lists all symbols included in the source program, with the following information:

- Name
- Address
- Type
- Usage
- Storage
- Source program references

1.7.1.1 Name, address, and type fields

The first three fields give the following information:

- **Name:** the symbol's name as represented in the symbol table. The field is as long as the longest symbolic constant name in the source program, but cannot exceed 31 characters; the minimum length is 8 characters.

- **Address:** a symbol's address, represented in 8 octal digits, is the offset of the variable's starting location in its corresponding storage type block. This field is blank if a storage location is not assigned to the symbol or if code is not generated for the program.

- **Data type:** blank if the symbol is not associated with a type.

1.7.1.2 Usage field

The usage field describes the symbol's usage in the program unit. If a symbolic name appears as a dummy argument, it is indicated in the storage field. Symbol usages are as follows:

Usage	Representation
Variable	**Var**
Array	**Array**
External function or subroutine	**External**
Entry	**Entry**
Statement function	**Stmt_func**
Intrinsic	**Intrinsic**
Array in equivalence	**Array,Eqv**
Parameter	**Parameter**
Variable in equivalence	**Var,Eqv**
Common block name	**Comm_blk**

1.7.1.3 Storage field

The storage field indicates the type of storage allocated to a symbol. If a symbol is assigned to a common storage block, this field represents the common block name corresponding to it. The corresponding address field represents the offset from the start of the common block. For symbols that have been assigned either static or stack storage type, the address field represents the offsets in their corresponding storage blocks. A blank field is printed if the variable is not assigned storage. (This occurs frequently for local variables when optimization is enabled.) Storage types and their representation are as follows:

Representation	Storage or Use	Description
Static	Static storage	Name resides in static memory
Stack	Stack storage	Name resides in stack memory
Pointee	Storage not assigned	Name applies to content of any address pointed to by the pointer
common block name	Common storage	Name resides in block static memory
Heap	Heap storage	Name resides in heap storage
Dum_arg	Dummy argument	Name is a dummy argument and does not have storage

1.7.1.4 Source program references

The source field shows source program line numbers where the symbol has appeared. Depending on the statement and the context, cross reference code is generated for each appearance of the symbol. The code uses the following format:

```
linenum xrefcode [/count]
```

linenum Line number of the statement (not the statement number). This number is counted from the beginning of the program unit.

xrefcode Cross-reference code. Codes and their meanings are as follows:

- **A** Appeared as an actual argument
- **D** Symbol is declared
- **M** Symbol's value may get changed or modified
- **U** Symbol's value is used

count Number of appearances in the same line; contains a blank if the variable has appeared only once in a line.

1.7.2 PARAMETER TABLE

The Parameter Table lists the names of symbolic constants in alphabetical order. It contains the following information for each symbol found in the PARAMETER statement of the source program.

- The **Name** column contains the name of the symbolic constant.
- The **Type** column shows the data type of the symbolic constant. Data types are represented as shown in 1.7.
- The **Value** column gives the actual value of each symbolic constant.

1.7.3 LABEL CROSS-REFERENCE TABLE

The Label Cross-reference Table consists of the following information for each label in the program.

- The **Label** column shows the statement number in the program.
- The **Defined** column shows the source program line number where the label is defined, counted from the beginning of the program unit. If a label is not defined in the program it is marked "undefined".
- The **Reference** list shows the source program line numbers where the label is referenced.

LANGUAGE ELEMENTS AND STRUCTURE 2

This section describes the elements of the Fortran language, the structure of a Fortran program, and the statements that determine a program's structure.

2.1 ELEMENTS OF THE FORTRAN LANGUAGE

Fortran is coded in *characters* to form *syntactic items*, which are used as elements in *statements*.

2.1.1 CHARACTER SET

CFT77 recognizes the following character set:

- The 49 characters specified by the ANSI standard: 26 capital letters, 10 digits, the space character, and the following *special characters*:

  = + - * / ( ) , . $ ' :

- The lowercase letters (non-ANSI).

- The quotation mark ("), exclamation point (!), underscore (_), and the tab character (ASCII HT). These characters are non-ANSI.

Appendix A shows the ASCII codes for these characters in octal, hexadecimal, and decimal. These values establish an order for the characters, called the *collating sequence*. In this sequence, digits precede letters, and capitals precede lowercase. An *alphanumeric character* is any letter or digit.

In a CFT77 program, a capital letter and the corresponding lowercase letter are considered to be the same except within character or Hollerith constants. Output listings show source programs as they are received, leaving case intact. Error messages use mixed case. Other listings, such as cross reference lists, use only uppercase, except within character or Hollerith constants, where no case conversion occurs.

The ANSI Fortran Standard does not include lowercase letters, double quotation mark, underscore, exclamation point, or tab.

The ANSI Fortran Standard does not specify a collating sequence except among capital letters (A through Z) and digits (0 through 9).

2.1.2 SYNTACTIC ITEMS

Syntactic items are sequences of characters recognized by the CFT77 compiler and used to form statements. Except within character and Hollerith constants, blanks are ignored. Syntactic items are as follows:

- Constant: an unvarying value. See section 4 concerning constants, variables, and arrays.
- Symbolic name: the name of an entity that can be any of several possible kinds (see 2.1.5).
- Statement label: one to five digits, at least one nonzero. Leading zeros are ignored. For example, 22, 022, and 2 2 are equivalent.
- Keyword: a sequence of letters having special significance in Fortran statements, such as INTEGER, WRITE, and GOTO. The context in which a character sequence appears affects how it is interpreted; for example, in GOTO 1, GOTO is a keyword; in GOTO=3.2, GOTO is a symbolic name.
- Operator: one or two special characters or a combination of special characters and letters, used to specify an operation. See section 5.
- Special character: characters used as operators; commas in lists; and parentheses to establish precedence in expressions. See section 5.

A *list* is a sequence of one or more syntactic items separated, if more than one, by commas. The syntactic items appearing in a list are called *list items*.

2.1.3 LINES

A line can contain up to 96 columns. Columns 73 through 96 are ignored by CFT77. A Fortran program is expressed as an ordered sequence of the following types of lines:

* Initial
* Continuation
* Terminal
* Comment
* Compiler directive

The ANSI Fortran Standard limits line length to 72 characters.

2.1.3.1 Initial and terminal lines

An *initial line* contains all of, or the first part of, a single Fortran statement in columns 7 through 72. Columns 1 through 5 can include a statement label of one to five digits or blanks; the label is not part of the statement. 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,* not to be confused with a DO-loop's terminal statement, is an initial line containing only the END statement (see 6.5.2).

2.1.3.2 Continuation lines

A *continuation line* extends an initial line for expressing a single Fortran statement; up to 19 can be used after an initial line, and comment lines can be interspersed. (The initial line of such a sequence must not be a terminal line.) A continuation line 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 be blank.

2.1.3.3 Comment lines and embedded comments

A *comment line* has the character C, *, or ! in column 1, or only blank characters in columns 1 through 72. (See compiler directive lines.) Comment lines are ignored.

An *embedded comment* is a comment on the same line with a Fortran statement. Comments can be embedded in any statement except a FORMAT statement. When an exclamation point (!) appears outside a character constant, the remainder of the line is treated as a comment. The exclamation point cannot appear in columns 2 through 5. An exclamation point in column 6 indicates continuation of the previous statement, not continuation of an embedded comment on the previous line.

Example:

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

The exclamation point in column 6 of the second line above causes the line to be a continuation line of statement 10. The other exclamation points denote embedded comments.

The ANSI Fortran Standard does not provide for embedded comments.

2.1.3.4 Compiler directive lines

A *compiler directive line* is a line having the characters CDIR$ in columns 1 through 5; it can contain one or more compiler directives, provided only one contains an argument list. The C in the first column causes other compilers to treat CFT77 compiler directive lines as comments (see 1.4).

The ANSI Fortran Standard does not provide for compiler directives.

2.1.4 STATEMENTS

A *statement* is a sequence of syntactic items specifying an operation to be performed, characteristics of data, or information about the program. Except in assignment statements and statement functions, the first syntactic item (following the statement label if any) in a statement is a keyword. A single statement comprises an initial line and up to 19 continuation lines. Blanks are ignored, except those within character or Hollerith constants. A label can precede a statement but is not a part of the statement.

The use of a statement is indicated by its keyword or by its form. A statement can have up to 1,320 characters including blanks. Aside from this limitation, blanks do not affect interpretation.

2.1.4.1 Kinds of statements

Executable statements specify actions and form an execution sequence. They are the following. (Groupings are informal: not ANSI categories.)

* Assignment:
  - = (arithmetic, logical, character)
  - ASSIGN (statement label)

* Program control:
  - DO
  - CALL, RETURN
  - Conditional block: Block IF (IF THEN), ELSEIF, ELSE, ENDIF
  - IF (arithmetic and logical)
  - GOTO (unconditional, assigned, computed)
  - CONTINUE, PAUSE, STOP, END

* Input/output:
  - Data transfer: READ, WRITE, PRINT, PUNCH
  - Sequential file positioning: REWIND, BACKSPACE, ENDFILE
  - File access control: OPEN, CLOSE
  - INQUIRE
  - BUFFER IN and BUFFER OUT
  - ENCODE and DECODE

Nonexecutable statements specify characteristics, arrangement, and initial values of data; specify editing information; specify statement functions; classify program units; and specify entry points within subprograms. Nonexecutable statements are not part of the execution sequence; they may be labeled but their labels cannot be used to control the execution sequence. Nonexecutable statements are the following. (Groupings are informal: not ANSI categories.)

* Program unit:
  - Header: PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA
  - ENTRY

* Specification:
  - Type declaration: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, POINTER
  - IMPLICIT, IMPLICIT NONE
  - PARAMETER, DATA, SAVE, EQUIVALENCE
  - COMMON, TASK COMMON, LOCAL COMMON
  - EXTERNAL, INTRINSIC
  - FORMAT, NAMELIST
  - Statement function definition statement
  - DIMENSION
  - INCLUDE

2.1.4.2 Order of statements and lines

The various kinds of statements and lines must appear in a specific order. Table 2-1 illustrates the required order of statements and lines for a program unit. The top-to-bottom order indicates the first-to-last appearance of lines and statements in a program unit source code. Statement order is shown as follows:

- Vertical lines divide 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 divide varieties of statements that must not be interspersed. For example, statement function definition statements must not be interspersed among executable statements.

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 2-1. Required Order of Lines and Statements

← Intermixing of statements is permitted →

Source statement order within a program unit ↓

<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, FORMAT, and DATA†† statements</td><td rowspan="2">PARAMETER statements†</td><td>IMPLICIT NONE and IMPLICIT†</td></tr>
<tr><td>Other specification statements</td></tr>
<tr><td colspan="2">Statement function definition statements</td></tr>
<tr><td colspan="2">Other executable statements</td></tr>
<tr><td colspan="4">END statement</td></tr>
</table>

† An IMPLICIT statement must precede a PARAMETER statement to affect the typing of constants named in the PARAMETER statement.

†† The ANSI Fortran standard specifies that DATA statements follow all specification statements.

2.1.5 SYMBOLIC NAMES

A *symbolic name* is the name of a constant, variable, array, common block, main program, external function, subroutine, intrinsic function, statement function, block data subprogram, dummy procedure, or namelist. Each of these uses is referred to as a *class*.

A *reference* (to a constant, variable, array, array element, or function) occurs when a symbolic name appears in a context where a value is required; such a reference is called a *constant reference*, etc. A reference to a variable, array, or array element provides the value currently associated with that entity; the value is not modified.

The *scope* of a symbolic name is the range, within a program, where the name can be used. Most names are either global or local:

- A *global* name has a scope of an entire executable program. Global names identify the main program, common blocks, subprograms, and external procedures. A global name must not exceed eight alphanumeric characters or underscores and must not have two different global uses in the same executable program.

- A *local* name's scope is a single program unit. A local name identifies an array, variable, constant, statement function, or intrinsic function. Local names must not exceed 31 alphanumeric characters or underscores, the first of which must be a letter. A name must not have two local uses in the same program unit and must not be the same as any global name except that of a common block (see 4.6.3).

- The names of variables that appear as dummy arguments in a statement function statement have a scope of only that statement.

- The name of a DO variable in an implied-DO list within a DATA statement has a scope of that list.

Notice that global and local names do not correspond to global and local data, which is discussed in 4.5.5.2. The use of the terms *global* and *local* for symbolic names is ANSI terminology; in Cray usage, the terms are often replaced by *external* and *internal*, respectively. A global name is external in the sense that it is used by system software, such as the loader.

The ANSI Fortran Standard provides for symbolic names of up to 6 alphanumeric characters.

Some character sequences, such as format edit descriptors and keywords that uniquely identify certain statements (GO TO, READ, FORMAT, and so on) are not symbolic names.

2.2 THE EXECUTABLE PROGRAM

An *executable program* consists of a group of one or more program units and procedures. A *program unit* (see 2.3) is an ordered set of Fortran statements, which can be a main program or a subprogram:

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

- A *subprogram* is any program unit other than the main program. It can be a *procedure subprogram,* which specifies a procedure, or a *specification (block data) subprogram,* which contains only nonexecutable statements (such as for initializing variables).

2.2.1 PROCEDURES: SUBROUTINES AND FUNCTIONS

A *procedure* is executable code, not necessarily Fortran, that can be called. That is, it can be invoked, from a program unit or another procedure, by a *procedure call.* A called procedure can process variables, and expressions containing variables, that are specified in a procedure call, creating new values for the use of the calling program unit. A call can appear at various points in either a program unit or another procedure.

A procedure can be a subroutine or a function:

- A *subroutine* (see 2.5.2) is invoked only by the CALL statement.

- A *function* (see 2.4 and 2.5.1) is invoked by a reference to it in an expression; the function reference is given a value, called the *function value.*

A function can be one of the following:

- A *statement function* (see 2.4.2) is specified by a single statement within the program unit that uses the function.

- An *intrinsic function* (see 2.4.3) is included with a given compiler and is always available with that compiler unless its name is used in another way.

- An *external function* (see 2.5.1) is specified by user-supplied code outside the calling program unit.

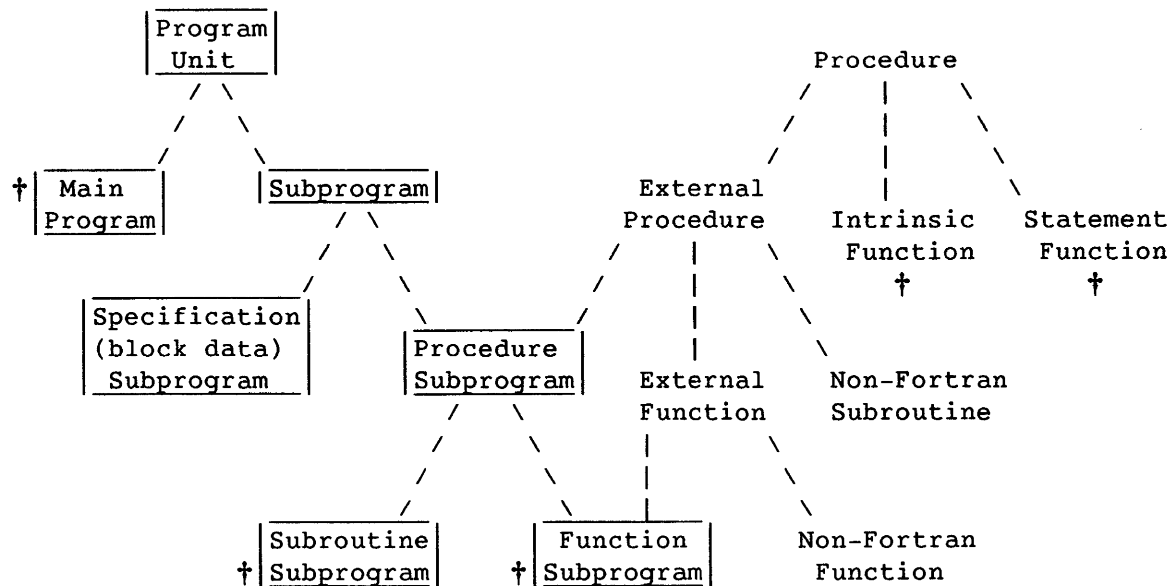
Code for specifying a procedure can be any of the following: a Fortran subprogram (function or subroutine), a statement within a program unit (statement function), code provided by the compiler (intrinsic function), or non-Fortran code provided by the user or the system (external function or subroutine).

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

2.2.2 SUMMARY OF PROGRAM STRUCTURE

The two primary categories of Fortran structure are the program unit and the procedure. A program unit can call, and a procedure can be called. A program unit must be coded in Fortran, whereas a procedure can be coded in any language. These categories overlap in the procedure subprogram, which is a Fortran entity that can both call and be called.

Figure 2-1 illustrates the breakdown of Fortran terms. All subcategories of a given term appear below the term, connected to it with dashed lines. All instances of program units are enclosed in boxes. The five instances of the term *function* and and the two instances of the term *subroutine* are not tied to unifying labels. Figure 2-2 shows a program that includes examples of the most important terms.



Boxes indicate examples of Fortran program units.

† An example of this term appears in figure 2-2.

Figure 2-1. Subcategories of Fortran Terms

Figure 2-2 shows a program that includes many of the Fortran concepts defined in the preceding discussion and in 2.2.3. It does not include programming concepts such as loops and arrays. Variable ARG is used as an actual argument for statement function STMTFUNCT, intrinsic function SQRT, and both subprograms. All dummy arguments begin with DUM. "C" indicates a comment line.

```
C  Main program reads value for ARG, computes value for variable
C  VARIABLE, prints value. The expression to compute VARIABLE uses
C  three kinds of functions:  statement function STMTFUNCT, function
C  subprogram FUNCTSUBP, and intrinsic function SQRT. Subroutine
C  subprogram SUBRTINE is called to check values.
C
      PROGRAM TERMS
      STMTFUNCT(DUMSF) = DUMSF * 10.0     ! Stmt function definition
      READ *, ARG
      CALL SUBRTINE (ARG)
      VARIABLE = STMTFUNCT(ARG) + FNCTSUBP(ARG) + SQRT(ARG)
      CALL SUBRTINE (VARIABLE)
      PRINT *, VARIABLE
      END
C
C
C  Subroutine subprogram checks for unreasonable values. DUMSR is
C  the dummy argument; it takes the value of ARG from the CALL
C  statement above.
C
      SUBROUTINE SUBRTINE (DUMSR)
      IF (DUMSR .GT. 10**8) STOP
      END
C
C
C  Function subprogram computes the function value FNCTSUBP
C  depending on the value of dummy argument DUMFS.
C
      FUNCTION FNCTSUBP (DUMFS)
      IF (DUMFS .LT. 51.0) THEN
        FUNCTSUBP = DUMFS + 3.2/DUMFS**2 + 15.6
      ELSE
        FUNCTSUBP = DUMFS + 15.6
      ENDIF
      END
```

Figure 2-2. Example Program Showing Fortran Structure

2.2.3 COMMUNICATING DATA WITHIN PROGRAMS

If a called procedure is to process data known to the calling program unit, entities (such as variables) in the procedure and program unit must be *associated;* that is, they must represent the same memory locations (see 4.5). This association is established by arguments and common blocks, discussed in the following paragraph, as well as by variables and arrays within a program unit. These methods can be used as follows:

Kind of Association	Arguments	Common Blocks	Variables or Arrays Within Program Unit
Subroutine or function subprogram, or other external procedure	X	X	
Statement function	X		X
Intrinsic function	X		

An *actual argument* is an entity specified in a procedure call; a *dummy argument* is an entity, specified in a procedure, that becomes associated with an actual argument used in a call to the procedure. If an actual argument is an expression, its value is given to a temporary variable, which is then associated with the called procedure's dummy argument. Figure 2-2 shows the use of arguments in calls to subprograms; arguments are discussed in detail in 2.6.

A *common block* is an area of memory that can be referenced by any program unit or procedure in a program. When the same common block is declared in two program units, corresponding entities in the common block are associated, and their values are available to both program units without the use of arguments. A *named* common block has a name specified in a COMMON statement, along with the names of variables or arrays stored in the block. See 4.6.

2.3 PROGRAM UNITS

A program unit contains a sequence of Fortran statements and optional comment lines. An executable program must include one main program and can include one or more subprograms.

The main program's first statement can be a PROGRAM statement. A subprogram begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement. Each program unit must end with an END statement. The main program can reference one or more subprograms during its execution; neither the main program nor a subprogram can reference a main program.

Execution of a subprogram can be begin or end at various points within the subprogram, specified by the ENTRY and RETURN statements (see 2.5.3). In static mode (the default), execution of a procedure subprogram must be terminated by a RETURN or END statement before the subprogram can be referenced again. In stack mode, CFT77 allows recursive use of procedures (non-ANSI; see 2.5). Static and stack are discussed in 4.5.2.

2.3.1 PROGRAM STATEMENT

Although the PROGRAM statement is optional, it is required for option **f**/F (Flowtrace; see 1.4.3); and option **h**/H (listing of all header statements; see tables 1-1 and 1-2). When used, it is the first statement of the main program.

PROGRAM *pgm* [(*h*)]

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

h Any sequence of allowed characters (see 2.1.1); has no effect on the executable program, but is allowed so that programs for other implementations of Fortran will run.

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

Example:

```
PROGRAM A1B2C3D4

PROGRAM X (INPUT,OUTPUT)
```

2.4 FUNCTIONS

A *function* is an executable entity invoked by a function reference (see 2.4.1) used as a primary in an expression. The reference becomes associated with a value, called the *function value*. Example:

```
Y = FUN(X) + 2.0
```

The above statement calls function FUN to process the current value of X. The function reference FUN(X) takes on a new value, which is then used in the right-side expression to assign a new value to Y.

A function is one of the following:

- A *statement function* is specified by a single statement within the program unit that uses the function (see 2.4.2).
- An *intrinsic function* is provided by CFT77 and is always available unless an external function of the same name is substituted by use of the EXTERNAL statement, or unless its name is used for another entity (see 2.4.3).
- An *external function* is specified by user-supplied code outside the calling program unit; if it is written in Fortran it is a *function subprogram*. (Routines in other languages are not called "subprograms" in ANSI terminology.) See 2.5.1.

2.4.1 FUNCTION REFERENCE

All functions use the same form of reference:

fun([*a*[,*a*]...])

fun Symbolic name of a function or dummy procedure

a Actual argument; the parentheses are required even with no argument. Requirements for function arguments are discussed in 2.4.2, 2.4.3.1, and 2.6.2.

The types of actual arguments specified in a function reference must agree with associated dummy arguments defined in the called function. In standard Fortran, the number of actual and dummy arguments must also agree, but this requirement is not enforced by CFT77. However, if a dummy argument that is not associated with an actual argument is referenced, an

error or undefined value will result. To prevent this, any dummy argument not associated with an actual argument must follow, in the dummy argument list, all dummy arguments that are associated with actual arguments.

Examples:

```
Intrinsic:         SIN(T)
                   LOG (X**2 + Y**3 - 1.53)
                   ATAN2(U,V)
                   MAX(I,J,K,L,M)

User-specified:    FUNCTSP(D+A/1.414)
                   STMTF(A,B)
```

Examples of function references within statements:

```
(1)   Y = SIN(T)

(2)   M = MIN((MOD(I,J)+3),14) - 7
```

Example 2 above shows nesting of functions and functions used within expressions.

2.4.1.1 Data type of a function: reference versus value

The name used to call a function has a type within the calling program unit. This type is distinct from the type of the value generated by the function. For valid results, these types must agree, but the agreement is not enforced in Fortran. The type of a function name is established in the same way as a variable's type, either implicitly or by means of a type or IMPLICIT statement (see 3.1). The type of the function value is established in the code that executes the function. Example:

In the calling program unit:

```
...
INTEGER FUN, V, C           ! Type declaration of function FUN
...
V = FUN(C)
```

In the called function subprogram:

```
INTEGER FUNCTION FUN(X)     ! Header declares fnct as integer type
INTEGER X                   ! X requires its own declaration
...
FUN = ...         ! Function value is defined here; integer type
```

The above subprogram generates a value whose type agrees with that of the name used to call the function.

Fortran does not check for type agreement between an external function reference and its function value; nor does it convert the types of function values, as is done for variables in expressions. A function value of the wrong type therefore causes invalid results. You are responsible for agreement between external function references and their values. See 2.4.3.1 and 2.6.4.2 concerning intrinsic functions.

In addition to the requirements for type agreement, functions of type character also require agreement in the length of the character value. The length of a character function reference must be declared by a previous CHARACTER or IMPLICIT statement (see examples in 3.7.1.2).

2.4.1.2 Execution of functions

Execution of a function reference results in the following actions.

1. Actual arguments that are expressions are evaluated.

2. Actual arguments are associated with dummy arguments.

3. The function is executed.

4. Control is returned to the calling program unit. The function value is then used in evaluating the expression that contains the function reference.

Executing 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 a common block that affects the value of any other function reference in the same statement (see 4.6). Example:

```
FUNCTION X(PX)
COMMON/C/A
...
A=2
X=...
END

FUNCTION Y(PY)
COMMON/C/A
...
PY = A
Y=...
END

PROGRAM P
Z = X(Z)+Y(J)           !Illegal statement
END
```

The next-to-last line above is illegal because function X changes a variable in a common block that function Y also references.

If a function reference in a statement causes an actual argument of the function to be defined, that argument or any associated entities (see 4.5.5) must not appear elsewhere in the same statement. Example:

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

Y=G(X)+X
```

The above statements, in which 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.

2.4.1.3 Order of evaluation

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

Examples:

(1)

```
IF (F(Y) .EQ. 1.0) A=F(Z)
```

In the above statement, where F is a function name, the function reference in the conditional statement A=F(Z) is evaluated last (but only if the IF expression is true).

(2)

```
A=F(G(X))
```

In the above statement, where F and G are functions, G is evaluated first.

2.4.2 STATEMENT FUNCTIONS

A statement function is a function specified by a *statement function definition statement*. This statement appears after the specification statements, before the first executable statement, and before any statement function definition that references the function; it is nonexecutable and is not part of the normal execution sequence. A statement function can be referenced only in the program unit that contains the statement function definition statement.

An actual argument in a statement function reference can be any expression except an array expression (see 4.3.10) or a character expression in which one operand's length is specified as (*) (unless the operand is defined in a PARAMETER statement).

A statement function definition statement in a function subprogram can reference that function subprogram. This is recursion and is permitted only if **-a stack** is on the **cft77** command or ALLOC=STACK is on the CFT77 control statement.

The ANSI Fortran Standard does not allow a statement function statement in a function subprogram to reference that subprogram.

2.4.2.1 Statement function definition statement

The statement function definition statement specifies a function for use within the same program unit. Its format is as follows:

fun ([*d*[,*d*]...]) = *e*

fun — Symbolic name of the statement function, from 1 to 31 alphanumeric characters; local to the program unit that contains the function. The name must not appear in any specification statement other than a type statement (to specify the type of the function). The name cannot be an actual argument, nor any local or global entity except a common block name in the same program unit.

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

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

The ANSI Fortran Standard specifies that symbolic names can have a maximum of six characters.

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

- As a dummy argument of the same type in another statement function definition statement
- As a variable of the same type appearing elsewhere in the program unit
- As a common block name
- As a dummy argument in a FUNCTION or SUBROUTINE statement in the same subprogram that contains the statement function statement

Each primary of 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, which can be used elsewhere in the same program unit
- An array element reference
- An intrinsic function reference
- A reference to a statement function defined previously in the same program unit
- An external function reference; this must not cause a dummy argument of the statement function to become undefined or redefined.
- A dummy procedure reference
- An expression enclosed in parentheses, subject to the same rules as the larger expression *e*.

Examples:

(1) Definition statement: `DISCRIM(X) = X**(2.32-X) + 1.5*X`

    Function in use: `P = DISCRIM(A)`

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

    Function in use: `DELTA = ROOT(Q,R,S,ON)`

(3)
```
CHARACTER*10 S, T*20
S(I) = T(I:I+9)        !Illegal: substrings not allowed
```

(4)
```
E(A,I) = A(I)          !Illegal: array name cannot be actual arg.
```

Expression *e* can contain variables that are not dummy arguments; and the statement function statement does not require a dummy argument. Therefore a statement function can generate a result that is derived exclusively from variables outside the statement function statement. Example (where R appears elsewhere in the same program unit):

```
VOL( )=4.1887901*R**3
```

If a name in expression *e* is the name of both a dummy argument and an entity outside the statement function statement, it applies to the dummy argument only, and is not a reference to the other entity. Example:

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

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

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

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

2.4.3 INTRINSIC FUNCTIONS

An *intrinsic function* is a prespecified function for performing common operations and is always available unless you replace it with an external function of the same name, or unless it is the name of a dummy argument, array, variable, statement function, or NAMELIST group.

Some intrinsic functions are called from libraries included with the CFT77 compiler, while others cause CFT77 to generate in-line code. Intrinsic functions contain optimized code and frequently run faster than user-supplied code. To replace an intrinsic function with a user-supplied function of the same name, use the EXTERNAL statement to declare the name an external function (see 2.6.4.1).

CFT77 intrinsic functions include the entire set specified in the ANSI Fortran Standard, as well as a set of extensions; they are listed in appendix B. Some utilities that are called as functions are described in the Programmer's Library Reference Manual, CRI publication SR-0113.

2.4.3.1 Referencing intrinsic functions

Intrinsic functions are referenced as shown in 2.4.1. Actual arguments for an intrinsic function must agree in type, number, and order with those shown in appendix B. An actual argument can be any expression (including an array expression; see 4.3.11.3), except a character expression in which an operand's length is specified as (*) (unless the operand is the symbolic name of a constant).

Generic function names are used for families of intrinsic functions that perform similar operations but differ in the data types required for arguments and generated as results. These names simplify referencing because the same function name can be used with more than one type of argument.

However, generic use of a function name used as an actual argument gives invalid results, which cannot be detected in Fortran (see 2.6.4 and 2.6.4.2). Appendix B lists the intrinsic functions in groups, with the first name in each group serving as a generic for the whole group.

Example:

```
REAL R
COMPLEX C
DOUBLE D
X = SQRT(R) * SQRT(C) * SQRT(D)
```

The assignment statement above invokes the real, complex, and double precision versions of function SQRT. It is equivalent to the following:

```
X = SQRT(R) * CSQRT(C) * DSQRT(D)
```

2.4.3.2 Restrictions

Intrinsic functions are undefined for some values such as LOG(-7.). Out-of-range arguments cause run-time messages to be issued. Appendix B shows allowable argument ranges.

Example:

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

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

2.5 SUBPROGRAMS

A *subprogram* is a program unit that is not the main program. A subprogram can be a *procedure subprogram,* which specifies a procedure (function or subroutine) and is invoked by another program unit, or a *specification (block data) subprogram,* which contains nonexecutable statements such as those for assigning initial values to entities in common blocks.

A procedure subprogram can be one of the following:

- A *subroutine subprogram* specifies a subroutine, which is a procedure called by a CALL statement.

- A *function subprogram* specifies a function (see 2.4).

Some subroutines and external functions are not written in Fortran (and are therefore not considered "subprograms" in ANSI terminology.) They can be written in Cray Assembly Language (CAL) or a high-level language, and are separately assembled or compiled. See appendix F and the Macros and Opdefs Reference Manual, CRI publication SR-0012.

The block data subprogram exists so that initialization of common blocks can be performed in a separate program unit. Because most requirements for subroutine and function subprograms do not apply to the block data subprogram, it is described with common blocks (see 4.6.6).

A subprogram begins with a *header statement* (FUNCTION, SUBROUTINE, or BLOCK DATA), and ends with an END statement. In a procedure subprogram, a RETURN statement allows the return of control to the calling program unit before the end of the subprogram. A RETURN or END statement terminates execution of a subprogram and returns control to the calling program unit.

Execution of a procedure subprogram normally begins with the first executable statement following the header statement, but the ENTRY statement allows execution to begin at points within the subprogram (see 2.5.3.1). The number of ENTRY statements in a program unit is not restricted.

For a subprogram to process data that is known to the calling program unit, variables in the calling and called program units must be associated; that is, they must represent the same memory locations. This association can be established by an argument list (see 2.6) or by common blocks (see 4.6).

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

- Entities in a common block
- Initially defined entities (that is, those defined in a DATA statement; see 4.5.3 and 4.4)
- Entities specified by SAVE statements (see 4.5.4)
- Other entities in static storage

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

In static mode (the default), function and subroutine subprograms may not reference themselves, either directly or indirectly. In stack mode a subroutine or function subprogram can reference itself; this is *recursion*. (Stack mode is specified by **-a stack** in the **cft77** command or ALLOC=STACK in the CFT77 control statement; see 4.5.2 concerning stack storage.)

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

2.5.1 EXTERNAL FUNCTIONS AND FUNCTION SUBPROGRAMS

An *external function* is a function (see 2.4) specified by user-supplied code that is external to the calling program unit; the code may or may not be a Fortran subprogram. A *function subprogram* is a Fortran subprogram that defines a function, and is one kind of external function. See 2.5 concerning non-Fortran functions.

2.5.1.1 Restrictions on external functions

If you wish to replace an intrinsic function with an external function, declare its name in an EXTERNAL statement (see 2.6.4.1). The loader determines which function is used.

An external function is not restricted to defining its function value. It can also define dummy arguments or entities in common blocks, so long as these redefinitions do not affect entities referenced in the same statement that references the function.

An external function name is global; it cannot be the same as any local name in a program unit where it is referenced or in the program unit that defines it, except as the function's result variable (see 2.5.1.2).

2.5.1.2 Function subprograms

A function subprogram begins with a FUNCTION statement and can contain any statement other than a BLOCK DATA, PROGRAM, SUBROUTINE, or another FUNCTION statement. A subprogram ends with an END statement.

A function subprogram normally executes from beginning to end; then control transfers to the executable statement (in another program unit) that called the function. This sequence can be altered in the following ways (see 2.5.3):

- The subprogram can begin executing at an ENTRY statement within the subprogram.
- A RETURN statement or STOP statement can end execution before the subprogram's END statement.

The symbolic name of a function subprogram and any entries (ENTRY statements) must appear as variable names in the function subprogram and must become defined during execution of the function; these variables are called *result variables* within a subprogram. The *function value* of a function reference is the value of the result variable of the same name when a RETURN or END statement is executed in the subprogram.

The symbolic name of a function specified by a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement except a type or EXTERNAL statement and must appear only as a variable or actual argument in executable statements; this restriction does not apply to recursive use of a function (CFT77 extension).

You must make sure that the function value generated by a function subprogram agrees with the type of the function reference in the calling program unit (see 2.4.1.1). The function value's type is specified in any of the following ways:

- The type is implicit in the function name.
- The type is specified in the FUNCTION statement.
- The type is declared by a type statement that includes the function name appearing in the subprogram.

If the type is specified in the FUNCTION statement, the function name must not also appear in a type statement; redundant type specifications are not allowed. If the function name is a character variable with a length specification of (*), it must not appear as an operand for concatenation except in a character assignment statement.

Each invocation of a function has its own function result variable, including recursive calls. See 2.5 concerning recursion.

In a function subprogram, the symbolic name of a dummy argument is local and must not appear in any of the following uses: in an EQUIVALENCE, INTRINSIC, PARAMETER, SAVE, or DATA statement; as a pointee; in a COMMON statement except as a common block name; or as a NAMELIST group name.

2.5.1.3 FUNCTION statement

A function subprogram begins with a FUNCTION statement, which identifies a subprogram as a function subprogram, establishes the function's symbolic name and, optionally, specifies its data type (see 2.4.1.1 concerning function data types).

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

type Declares the function's data type, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[**len*]. If the type is not declared in the FUNCTION statement, it can be declared in a type statement within the subprogram of the form *type fun* as in REAL NUM in subprogram NUM. Otherwise, the type is implicit in the function name.

> *len* Length of the result of a character function; can be an unsigned positive integer constant or a positive integer constant expression enclosed in parentheses (expression cannot include the symbolic name of a constant), or (*). See 3.7 concerning character type and 3.7.1.1 concerning the use of the asterisk specification.

fun Symbolic name of the function subprogram; global name of 1 to 8 alphanumeric characters. *fun* must appear as a variable name and become defined in the function subprogram. *fun* must not appear in any other nonexecutable statement other than a type statement or EXTERNAL statement, except as a common block name. *fun* must appear only as a variable or dummy argument in executable statements, unless the function is to be used recursively (non-ANSI; see 2.5).

d Dummy argument representing a variable, array, or dummy procedure name. Parentheses are required even with no arguments listed. See 2.6 concerning arguments.

Examples:

(1)

```
      FUNCTION MOM(N,L)
```

Because of implicit typing, a subprogram beginning with the above statement generates an integer value unless its type is changed by a type statement or IMPLICIT statement within the subprogram. A reference to this function could be of the form I=MOM(J,K). In the calling program unit, the function reference would be implicitly typed integer unless declared otherwise (see 2.4).

(2)

```
      REAL FUNCTION MOM(X,Y)
```

A function defined by a subprogram beginning with the above statement is explicitly typed real, so the value generated is real for a function call such as X=MOM(P,Q). The function reference would then have to be declared real in the calling program unit.

(3)

```
      FUNCTION AVERAGE(A,N)
      REAL A(N)
      SUM = 0.0
      DO 10 I=1,N
         SUM = SUM + A(I)
   10 CONTINUE
      AVERAGE = SUM/N
      END
```

The above function subprogram uses a dummy array and a DO loop to compute an average of the N values contained in an actual array. See 4.3 concerning arrays and 6.3 concerning DO loops. Notice that the function name is defined within the subprogram.

2.5.2 SUBROUTINES AND SUBROUTINE SUBPROGRAMS

A *subroutine* is an executable entity that is invoked only by a CALL statement within a program unit or other procedure. A subroutine is distinct from a function in that it is called only by the CALL statement and has no data value associated with its name. A subroutine name cannot be used in an expression, and it has no data type.

A subroutine can be specified by either a Fortran subprogram or a non-Fortran procedure (described in appendix F). A Fortran subprogram that specifies a subroutine is a *subroutine subprogram*.

Users of Cray computer systems have access to a library of subroutines for a variety of purposes, including mathematical, scientific, and system utilities. These are described in the Programmer's Library Reference Manual, CRI publication SR-0113.

If a subroutine is to process data known to the calling program unit, variables in the subroutine and program unit must be associated; that is, they must represent the same memory locations. This association can be established by an argument list (see 2.6) or by common blocks (see 4.6).

A subroutine subprogram normally executes from beginning to end; then control transfers to the statement following the CALL statement that called the subroutine. This sequence can be altered in the following ways (see 2.5.3):

- The subprogram can begin executing at any point that an ENTRY statement appears.
- The RETURN statement can end execution before the end.
- The RETURN statement can specify a different point in the calling program unit to resume execution. This is an *alternate return*.

2.5.2.1 Requirements

A subroutine subprogram begins with a SUBROUTINE statement containing the subroutine's name and the names of any dummy arguments or alternate return specifiers, and ends with an END statement. A subroutine subprogram cannot contain a BLOCK DATA, FUNCTION, or PROGRAM statement, or a second SUBROUTINE statement.

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

2.5.2.2 CALL statement (subroutine reference)

The CALL statement causes execution of the subroutine specified in the statement. It can specify actual arguments to be associated with dummy arguments in the subroutine, to allow the subroutine to process data as needed by the calling program unit. The CALL statement can also specify different statements to which control can be returned.

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

sub Symbolic name of a subroutine, subroutine entry (see 2.5.3.1), or dummy subroutine. *sub* can be a dummy subroutine name only within a subprogram, one of whose dummy arguments is *sub* (see 2.6.4).

a Actual argument or alternate return specifier. Permitted actual arguments are discussed in 2.6.2. An alternate return specifier (**s*, denoting a statement label) allows control to be transferred to a different statement in the calling program unit (see the RETURN statement, 2.5.3.2).

The types of actual arguments specified in a CALL statement must agree with associated dummy arguments defined in the called subroutine. In standard Fortran, the number of actual and dummy arguments must also agree, but this requirement is not enforced by CFT77. However, if a dummy argument that is not associated with an actual argument is referenced, an error or undefined value will result. To prevent this, any dummy argument not associated with an actual argument must follow, in the dummy argument list, all dummy arguments that are associated with actual arguments.

Examples:

```
CALL SAM
CALL GEORGE(X,-7.)
CALL TOM(*10,X,*20,Y)
```

Corresponding to the examples shown for the SUBROUTINE statement (2.5.2.3, following), the first example above calls subroutine SAM; the second calls subroutine GEORGE with actual arguments X and -7.; and the third calls subroutine TOM with actual arguments X and Y and alternate return specifiers for statements 10 and 20.

Execution of a CALL statement results in the following:

- Evaluation of actual arguments that are expressions (see 2.6.2)
- Association of actual arguments with the corresponding dummy arguments, as described in 2.6.2.
- The actions specified by the referenced subroutine

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

2.5.2.3 SUBROUTINE statement

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

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

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

d Dummy argument or asterisk. A dummy argument represents an entity used in the subprogram, corresponding to an actual argument in the CALL statement that calls the subprogram. An asterisk corresponds to an alternate return specifier in the CALL statement (see CALL statement, 2.5.2.2 and RETURN statement, 2.5.3.2).

Examples:

```
SUBROUTINE SAM
SUBROUTINE GEORGE(A,B)
SUBROUTINE TOM(*,X,*,Y)
```

The first example is the first statement of subroutine SAM, which has no dummy arguments or alternate returns. The second example is for a subroutine with dummy arguments A and B. The third is for a subroutine with dummy arguments X and Y, and two asterisks corresponding to alternate return specifiers in the CALL statement. The examples shown for the CALL statement (2.5.2.2, previous) correspond to these examples.

2.5.3 ALTERING THE TRANSFER OF CONTROL BETWEEN PROGRAM UNITS

The following statements let you choose, with some restrictions, where execution of a subprogram begins and ends, and to choose where execution resumes in the calling program unit after execution of a subroutine.

2.5.3.1 ENTRY statement

The ENTRY statement is used in a procedure subprogram to allow execution to begin at a point within the subprogram's execution sequence, rather than at the beginning. In the procedure call, the name in the ENTRY statement is used instead of the name appearing in the subprogram's FUNCTION or SUBROUTINE statement.

The ENTRY statement is nonexecutable and can appear anywhere in a procedure subprogram after the FUNCTION or SUBROUTINE statement, except within a DO-loop or IF structure (see 6.1). Execution begins at the ENTRY statement. A procedure subprogram can contain one or more ENTRY statements following its FUNCTION or SUBROUTINE statement.

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

en Entry name; this name is a subroutine name or function name used in procedure calls in the normal way. Restrictions are listed in a subsequent paragraph.

d Dummy argument representing a variable name, array name, procedure name, or an asterisk associated with an alternate return specifier (only in a subroutine subprogram). 2.6.2 discusses requirements for dummy and actual arguments.

Dummy arguments in an ENTRY statement need not agree with those specified in a FUNCTION, SUBROUTINE, or other ENTRY statement in the same subprogram. Any dummy argument must be named in an ENTRY statement or header statement before it appears in an executable statement or statement function statement.

Restrictions on the entry name *en* are as follows:

- *en* cannot be used as a dummy argument in the same subprogram.
- In a function subprogram, *en* must not appear as a variable in any statement preceding the ENTRY statement, except a type statement.
- *en* cannot be the same as any global name in the same executable program.
- *en* cannot appear in a POINTER or NAMELIST statement.
- In a function subprogram of type character, *en* must be of type character, with the same declared length as the function name, either an integer or (*) to denote a value specified in the function reference.

In a function subprogram, the function name *en* specified in an ENTRY statement can appear as a result variable within the subprogram. Even if it does not appear, it is associated with all other result variables within the subprogram, including the function name shown in the FUNCTION statement. Rules of association apply as described in 4.5. Entry names can differ in type from the name in the FUNCTION statement, except when the type is character.

Example:

```
PROGRAM ENTRYEX              REAL FUNCTION MEAN (ARRX)
REAL MEAN                    ...
...                          ENTRY MEDIAN (ARRX)
A=MEAN(ARR1)                 ...
...                          MEAN = ...
J=MEDIAN(ARR2)               IF (...) RETURN
...                          ...
END                          MEDIAN = ...
                             ...
                             END
```

In this example, because the functions MEAN and MEDIAN use some of the same code, they are combined into one function subprogram. The entry name MEDIAN is used as a function reference in the calling program unit and as a result variable in the subprogram. Note that MEAN is real and MEDIAN is integer type.

Because all result variables are associated, a conditional RETURN statement is needed so that the value for function reference MEAN is not changed by the assignment to MEDIAN. A change to MEAN would give at least an unintended result in the calling program unit, and in this case would create an undefined value because the two variables are of different types. If MEDIAN and MEAN were of the same type, the assignment to MEDIAN could instead be to MEAN, with the same result, because they are associated.

2.5.3.2 RETURN statement

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

In a function subprogram, the statement consists of only the word RETURN. In a subroutine subprogram, the format is as follows:

```
RETURN [i]
```

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

An *alternate return* allows a subroutine to return control to a statement identified in the CALL statement that called the subroutine. *i* in the RETURN statement references the *i*th asterisk in the dummy argument list of a SUBROUTINE statement; this asterisk in turn specifies a statement label preceded by the *i*th asterisk in the CALL statement. If *i* is less than 1 or greater than the number of asterisks specified, RETURN *i* is treated as RETURN.

Example:

```
      PROGRAM ALTRET
      ...
      CALL SUB (*5,A,B,*6)
    4 statement
      ...
    5 statement
      ...
    6 statement
      ...
      END
```

```
      SUBROUTINE SUB (*,S,T,*)
      ...
   10 RETURN 1
      ...
   11 RETURN 2
      ...
   12 RETURN I
      ...
   13 RETURN
```

Statement 10 returns control to statement 5 because RETURN 1 refers to the first alternate return specifier in the SUBROUTINE statement and therefore the first specifier (*5) in the CALL statement.

Statement 11 returns control to statement 6 because the second asterisk in the SUBROUTINE statement corresponds to the second specifier (*6) in the CALL statement.

Statement 12 returns control to statement 5 or 6 if I has a value of 1 or 2, respectively; otherwise control returns to statement 4.

Statement 13 returns control to statement 4.

2.6 ARGUMENTS

An *actual argument* is an entity, appearing in a procedure call, which becomes associated with a name in the called procedure; the name in the called procedure is a *dummy argument*. When the procedure is executed, any new value of the dummy argument is also the new value of the actual argument in the calling program unit.

Each procedure has a dummy argument list corresponding to the actual argument list in the procedure call; for example, in a subprogram the dummy argument list appears in the header statement. The two names are associated by their corresponding positions in the argument lists. See 3.7.3 concerning character arguments and 4.3.7 concerning array arguments.

2.6.1 ASSOCIATION OF ARGUMENTS

A procedure call associates actual arguments with dummy arguments: the first actual argument with the first dummy argument, and so on. The association is valid only if the arguments are of the same type. A dummy argument is undefined if it is not associated with an actual argument. Association of actual and dummy arguments ends when the procedure finishes executing; association is not continued to the next execution of the procedure.

The following requirements apply to the use of arguments:

- Dummy procedures and nesting: When a procedure name is used as an argument, its corresponding dummy argument must be a dummy procedure name (see 2.6.4). Association of arguments can be carried through more than one level of procedure reference; any invalid association makes further associations also invalid.

- Alternate return specifiers must be associated with asterisk dummy arguments (see 2.5.3.2).

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

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

- Common block entities: If a subprogram call associates a dummy argument with an entity in a common block, and the common block is declared or referenced in the subprogram, neither the common block entity nor the argument can become defined within the subprogram.

Example:

In calling program unit:	In called subprogram:
COMMON /CB/B	SUBROUTINE XYZ (A)
CALL XYZ (B)	COMMON /CB/C

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

2.6.2 ACTUAL ARGUMENTS FOR EXTERNAL PROCEDURES

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

The actual arguments in a reference to an external procedure must agree in order, number, and type with the dummy arguments in the procedure or procedure entry. CFT77 does not enforce agreement in number, and you can omit later arguments if you know that the corresponding dummy arguments will not be referenced (non-ANSI). An actual argument can be a subroutine name or an alternate return specifier (see 2.5.3.2); these do not have data types, so the requirement for type agreement does not apply.

An actual argument in an external procedure reference must be one of the following. (See 2.4.3.1 and 2.4.2 concerning actual arguments for intrinsic functions and statement functions.)

- Expressions: Any expression other than a) an array syntax expression or b) a character expression with an operand whose length is specified as (*), unless the operand is the name of a constant (see 3.7.3). If the expression includes an operator or parentheses, it is evaluated and stored in a temporary variable, which is then associated with the dummy argument. The following kinds of expressions involve these considerations:

  - Array element or substring name. The array subscript or substring designator is evaluated just before the arguments are associated, and it remains constant as long as the arguments are associated, even if the subscript or designator contains variables that are redefined during the association.

  - Function reference has its own argument list (empty where appropriate). The function is evaluated, and the function result is associated with the dummy argument.

- Procedure name. The name of an intrinsic function, external function, or subroutine. This allows a called procedure to call a second procedure that is specified in the first call (see 2.6.4).

- Array name. The referenced actual array must be at least as large as the dummy array in the called procedure (see 4.3.6.1 and 4.3.7).

- Alternate return specifier (subroutines only): A specifier of the form *n, where *n* is a statement label in the calling program unit (see 2.5.3.2).

- Dummy argument within the same subprogram that is making the procedure call. Example:

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

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

2.6.3 DUMMY ARGUMENTS

Subprograms use dummy arguments to indicate the types of actual arguments and whether each is a single value, an array, or a procedure. A reference to a subprogram causes its dummy arguments to become defined if the corresponding actual arguments are defined. A dummy argument name is local to a subprogram. Dummy arguments have the following requirements:

- Specify before using: If a dummy argument is referenced in an executable statement or statement function statement, it must be specified within the subprogram's header statement or a preceding ENTRY statement; or as a dummy argument within the same statement function statement.

- Kinds of dummy arguments: Each dummy argument is classified as a variable, array, or procedure. A dummy argument that is a variable can be associated with an actual argument that is a variable, array element, substring, or expression.

- Prohibited statements: 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 a common block name).

- Name duplication: A dummy argument name in a subprogram must not be the subprogram's name or the name of a statement function in the subprogram.

- Dummy array declarators: Adjustable dimension declarators can contain dummy arguments of type integer.

- Statement function dummy arguments must have scalar values (see 4.2).

2.6.4 DUMMY PROCEDURES

A *dummy procedure* is a dummy argument used as a procedure name in a call to an external procedure. This allows a procedure call to specify a another procedure to be referenced by the called procedure.

Example:

```
EXTERNAL AFUN                      SUBROUTINE SUB (D,DFUN)
...                                ...
CALL SUB (A,AFUN)                  VAR = DFUN(D)
...                                END
```

Above, dummy argument DFUN is associated with function name AFUN, which causes the function reference in the subroutine to reference function AFUN. Therefore DFUN(D) is equivalent to AFUN(A).

Before being used as an actual argument, a procedure name must be declared in an EXTERNAL or INTRINSIC statement, to distinguish it from other kinds of arguments. Statement functions cannot be used as dummy procedures. The names of intrinsic functions for lexical relation, maximum/minimum, and type conversion cannot be used as dummy arguments.

A dummy procedure name must be associated with an actual argument that is a procedure name. A subroutine name, dummy or actual, must not appear in a type statement or be referenced as a function.

The arguments for a dummy procedure must agree in number and type with those specified for an actual procedure with which it becomes associated. When a dummy procedure is a function, the dummy function reference must match the type of the actual function reference. A name appearing in a type statement and an EXTERNAL statement must be the name of a function.

2.6.4.1 EXTERNAL statement

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

- When an external or dummy procedure name is to be passed as an actual argument, even if the use of the name is not ambiguous. The EXTERNAL statement appears in the same program unit as the name's use as an actual argument.
- When you want to replace an intrinsic function with an external function of the same name. The intrinsic function of the same name is then unavailable, and the name becomes the name of an external function, either a function subprogram or a non-Fortran function.

EXTERNAL *proc* [,*proc*]...

proc Name of an external procedure, dummy procedure, or block data subprogram. Except for recursion (non-ANSI), *proc* cannot be an entry name into the same subprogram that contains the EXTERNAL statement. (Though block data subprograms can be declared external, they cannot be referenced or passed as arguments.)

A statement function name must not appear in an EXTERNAL statement. A given symbolic name can appear only once in all of the EXTERNAL statements of a program unit.

2.6.4.2 INTRINSIC statement

An INTRINSIC statement identifies a symbolic name as an intrinsic function. If an intrinsic function name is an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit.

INTRINSIC *fun*[,*fun*]...

fun Intrinsic function name

Some intrinsic function names cannot be used as actual arguments, though they are allowed in an INTRINSIC statement. These functions are those for lexical relation, maximum/minimum, type conversion, and vectorization, and names that are generic but not specific function names. These categories include the following functions:

AMAX0	CVMGT	IFIX	MAX
AMAX1	CVMGZ	INT	MAX0
AMIN0	DBLE	LGE	MAX1
AMIN1	DMAX1	LGT	MIN
CHAR	DMIN1	LLE	MIN0
CMPLX	FLOAT	LLT	MIN1
CVMGM	ICHAR	LOG	REAL
CVMGN	IDINT	LOG10	SNGL
CVMGP			

The appearance, in an INTRINSIC statement, of a generic function name that is also a specific name does not cause the name to lose its generic property when used in the same program unit. When the function name is passed as an actual argument, it is not generic when referenced in the called program unit. (This is because argument types cannot be known when the called program unit is compiled, and the function is accessed only by its address.)

Example:

```
INSTRINSIC SQRT
CALL JOE (SQRT)
...
SUBROUTINE JOE (F)
X = F(3.0,4.0)          ! Invalid result: CSQRT is not called
```

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.

DATA TYPES 3

Data can be specified in a Fortran program as a constant, a variable, an array, an array element, or a function reference. A *constant* is an invariant value, which cannot be modified. A *variable* is a name that can assume different values during program execution. An *array* is an ordered set of data items of the same type, identified by a single name. An *array element* is one item in an array and, like a variable, can assume different values during program execution; it is identified by the array name and by one or more expressions to specify its position within the array. See 2.4 concerning functions and section 4 concerning other terms above.

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

- Integer - integral, signed values
- Real - values approximating real numbers, consisting of a mantissa and an exponent
- Double-precision - values that are the same as real values, but extended to about twice the precision
- Complex - values approximating complex numbers as pairs of real data items. The first item in the pair represents the real portion and the second, the imaginary portion of the data.
- Logical - the logical values true and false
- Character - sequences of characters
- Pointer - values representing storage addresses; only a variable can be of type pointer. CFT77 extension.
- Boolean - values representing the binary contents of Cray words; only constants (octal, hexadecimal, or Hollerith), intrinsic functions, or expressions can be Boolean. CFT77 extension.

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

Table 3-1. Values Represented in Different Data Types

Value	Int.	Real	Double Precision	Complex
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.) (6920E-1,0.) (.692E3,0.) (6.92E2,0.)
6.128547472	6†	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.)
.875i	0†	0.†	0.0D†	(0.,.875) (0.,875E-3) (0.,.875E0) (0.,8.75E-1) (0.,.000000875E6)
692+.875i	692†	692.†	692.0D0†	(692.,.875) (692E0,0.875) (69.2E1,875E-3) (.692E3,875.E-3) (6.92E2,8.75E-1)

† This value differs significantly from the value in the leftmost column, but results from normal type conversion of that value.

3.1 TYPE SPECIFICATION

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

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

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

A function's data type must, in effect, be specified twice; the two specifications must agree for the function's result to be valid. The specifications are as follows:

- The type of the function's result value is determined in the coding of the function (see 2.4.2.1, 2.4.3.1, and 2.5.1.2).
- The type of the function reference in the calling program unit is specified in the same way as a variable's type; the type is either implicit in its name or declared within the same program unit (see 2.4.1 and 2.4.1.1).

3.1.1 TYPE STATEMENTS

A type statement declares the type of an entity, thereby either overriding or confirming the entity's implicit type, and can specify array dimensions by including array declarators with array names.

The appearance of a 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. A name's type must not be explicitly specified more than once within a program unit.

The name of a subroutine, main program, or block data subprogram must not appear in a type statement.

If a specific intrinsic function name appears in a type statement that conflicts with that function's type (as specified in appendix B), and the name is used as an intrinsic function name, the conflicting type statement is ignored and a warning message is issued. Note the requirements for function typing in the 3.1 introduction preceding.

The form of type statements other than POINTER and CHARACTER is as follows:

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

type Specifies type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL. (CHARACTER and POINTER type statements have different requirements; see 3.7.1 and 3.9.1. Boolean type does not have a type statement.)

v Symbolic name of a constant, variable name, array name, function name, dummy procedure name, or array declarator. An array declarator includes dimension declarators (see 4.3.4).

Examples:

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

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

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

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

In the above examples, numbers in parentheses are dimension declarators, and the names preceding the parentheses are array names. Other names can represent entities as listed in the preceding format description. See E.5 for extensions of the type declaration statements.

3.1.2 IMPLICIT STATEMENT

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

```
IMPLICIT type (a1[-a2][,a3[-a4]]...)[,type(a5[-a6][,a7[-a8]]...)]...
```

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

  len Length of the character entities. *len* can be an unsigned, nonzero, positive integer constant or expression; maximum values are shown in 3.7.

a Letter; a range of letters within the alphabet can be written in the form a_{first}-a_{last}; for example, I-M.

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

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

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

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

Examples:

```
IMPLICIT LOGICAL(L)
IMPLICIT DOUBLE PRECISION(X,Y),COMPLEX(C)
IMPLICIT INTEGER(A,B,F-K),REAL(M-W,Z)
```

The last example declares variables starting with A, B, F, G, H, I, J, and K to be integer type, and variables starting with M, N, O, P, Q, R, S, T, U, V, W, and Z to be real type.

3.1.3 IMPLICIT NONE STATEMENT (CFT77 EXTENSION)

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

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

When IMPLICIT NONE is specified, failure to provide type or EXTERNAL declarations is a fatal error for the cases already described. Intrinsic subroutine and function names need not appear in explicit type statements and must not be declared EXTERNAL.

3.2 INTEGER TYPE

An *integer* value represents positive, negative, or zero whole-number values with no fractional part. The form of an *integer constant* is an optional sign followed by a nonempty sequence of digits specifying a decimal integer value. An integer value occupies one storage unit in a storage sequence (see 4.5.1). Machine representation is shown in G.1.

The CFT77 default for internal representation of integers is 46 bits. With this default, integer values (I) can be in the following range:

$$-2^{46} \leq I < 2^{46} \text{ or (approximately) } -10^{14} < I < 10^{14}$$

The use of 64-bit integers can be specified by **-i64** in the **cft77** command, INTEGER=64 in the CFT77 control statement, or CDIR$ INTEGER=64 in your program. When this option is active, integer values can be represented in the following range:

$$-2^{63} \leq I < 2^{63} \text{ or (approximately) } -10^{19} < I < 10^{19}$$

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

3.3 REAL TYPE

A *real* value approximates the value of a real number. A real value occupies one storage unit in a storage sequence (see 4.5.1). Machine representation is shown in G.2.

A real constant is written as one of the following:

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

A *basic real constant* consists of an optional sign, an integer portion, a decimal point, and a fractional portion, in that order. The integer and fractional portions are sequences of digits representing integer and fractional decimal constants. Either, but not both, of these portions can be omitted. A basic real constant can be written with more digits than can be used to approximate its constant; the excess digits are lost by CFT77 in roundoff. Examples:

```
692.        692.0        34.836        0.458        .458
```

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

```
692.E0             692.0E0             6920E-1
```

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

$2^{-8193} \leq R < 2^{8191}$ or (approximately) $10^{-2467} < R < 10^{2465}$

The above numbers do not represent the range of values that can be used in all operations. For example, X=2.**8190 causes an error, but X=2.**8189 does not.

Nonzero real values have a maximum of 48 significant binary digits, or approximately 15 decimal digits of precision. Rounding, cancellation, and truncation can cause fewer than 48 reliable bits to be generated.

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

3.4 DOUBLE-PRECISION TYPE

A *double-precision* value is a signed approximation of a real number, extended to approximately twice the precision of a real value. Double-precision values can be positive, negative, or zero, and occupy two consecutive units in a storage sequence (see 4.5.1). Machine representation is shown in G.3.

Because the precision of CFT77's real type is comparable to double precision on many other systems, and because double precision slows execution, **-dp** on the **cft77** command and OFF=P on the CFT77 control statement are provided to cause double-precision values to be treated as single-precision real values. See tables 1-1 and 1-2.

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

- Basic real constant (see 3.3) followed by a double-precision exponent
- Integer constant (see 3.2) followed by a double-precision exponent.

A double-precision exponent has the same range as a real exponent. The form of a double-precision exponent is the letter D followed by an optionally signed integer value; the value preceding the exponent is multiplied by 10 to the specified integer value. Examples:

```
692D0            .692D3            612.8547472D-2
```

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

$$2^{-8193} \leq D \leq 2^{8191} \text{ or (approximately) } 10^{-2466} < D < 10^{2465}$$

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

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

3.5 COMPLEX TYPE

A *complex* value approximates a complex number as a pair of real values. The first item in the pair represents the real portion and the second, the imaginary portion of the value. A complex constant is written as a pair of integer or real constants, within parentheses and separated by a comma. A complex value occupies two consecutive storage units in a storage sequence (see 4.5.1): the first for the real portion and the second for the imaginary portion. Machine representation is shown in G.4.

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

$$2^{-8193} \leq C_{real} \text{ or } C_{imag} < 2^{8191}$$

or (approximately)

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

Each component contains a maximum of 48 significant binary digits, or approximately 15 decimal digits of accuracy.

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

3.6 LOGICAL TYPE

A *logical* entity is a single true or false value (occupying one storage unit), which cannot be used as a number. Variables, arrays, and constants can be of logical type. Logical entities are used in tests for conditional code (see 6.1). The difference between logical and Boolean types is discussed in 3.8.

A logical or relational expression results in a single logical value (or one value per element in an array expression) and can therefore be used in a conditional test. Logical expressions use logical operands, but relational expressions do not (see 5.4 and 5.3).

Logical values are represented as .TRUE. or .T. for true, and .FALSE. or .F. for false. Machine representation is shown in G.6.

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

3.7 CHARACTER TYPE

A *character* value or *character string* consists of one or more characters, as listed in appendix A. See 5.2 concerning character expressions and character assignment statements. Machine representation is shown in G.5.

A character constant is written as a sequence of characters preceded and followed by a delimiter, which can be an apostrophe or quotation mark. The delimiter can appear as a character within the string if it appears twice in succession; the double character is interpreted as a single character. Blanks in a character string are significant. Examples:

- 'ABC' or "ABC" represents ABC.
- '''' or "'" represents '.
- '""' represents "".

Each character within a string has a position that is numbered ordinally from the first character. These positions are used in specifying character substrings (see 3.7.2).

The length of a character constant is the number of characters between its delimiters (from 1 to 1316), with each pair of consecutive delimiters counted as a single character.

A variable or user-specified function of type character is declared with its length by the CHARACTER type statement. The bit length of a character dummy argument must be less than or equal to the length of the corresponding actual argument (see 3.7.3). A character value must have at least 1 character, and, unless further restricted by machine memory, fewer than 8,388,608 (2^{23}) characters on the CRAY-2 system or fewer than 2,147,483,648 (2^{31}) characters on other Cray systems.

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

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

3.7.1 CHARACTER TYPE STATEMENT

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

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

len — Length specifier (number of characters): an unsigned, nonzero integer constant or a positive, nonzero integer constant expression enclosed in parentheses. *len* can follow one name or can follow the word CHARACTER to apply to all names lacking length specifiers. If *len* does not follow CHARACTER, it defaults to 1. *len* must be less than 2^{23} on CRAY-2 systems and 2^{31} on other Cray systems. It can also be in the form (*); see 3.7.1.1.

name — Variable name, symbolic name of a constant, function name, dummy procedure name, array name, or array declarator (see 4.3.5).

Examples:

(1)

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

The above declares character variables X and Z of length 5, W of length 2, and Y of length 7.

(2)

```
CHARACTER ARR(3,5)*7,X
```

The above declares a 3-by-5 array ARR, each of whose elements contains seven characters, and character variable X of length 1.

3.7.1.1 Asterisk specification

len can be specified as (*) if *name* is the name of an external function, dummy argument, or character constant. The asterisk specification is treated as follows:

- If *name* is the name of a FUNCTION or an ENTRY statement in the same subprogram, the length is obtained from the calling program unit. See 3.7.1.2.

- If *name* is a dummy argument, the dummy argument assumes the length of the associated actual argument (see 3.7.3 concerning character arguments).

- If *name* is a character constant with a symbolic name, the constant will assume the length of its corresponding constant expression defined later in a PARAMETER statement.

Example:

```
CHARACTER DUMARG1*(*)
```

This statement declares a dummy argument with the asterisk length specification.

3.7.1.2 Character function declaration

When you use a character function, the function reference and function value must agree not only in type (see 2.4.1.1) but also in length. Use CHARACTER*(*) in a function subprogram when the length of the subprogram's result variable is to be determined by the length of the function reference in the calling program unit. The length of a dummy argument can be declared in the same manner, so that it has the same length as the associated actual argument.

Example:

In the calling program unit:

```
CHARACTER*15 LOWRCASE, UPTITLE, TITLE   ! Fnct reference declared
...
TITLE = LOWRCASE (UPTITLE)
```

In the called function subprogram LOWRCASE:

- The length of result value LOWRCASE can be declared in the following ways:

```
CHARACTER*(*) FUNCTION LOWRCASE (STRING)

FUNCTION LOWRCASE(STRING)
CHARACTER*(*) LOWRCASE

CHARACTER*15 FUNCTION LOWRCASE (STRING)

FUNCTION LOWRCASE(STRING)
CHARACTER*15 LOWRCASE
```

- In each case just shown, dummy variable STRING can be declared within the subprogram in the following ways:

  ```
  CHARACTER*(*) STRING

  CHARACTER*15 STRING
  ```

If a returned character value is longer than the function reference, the rightmost characters are truncated. If the returned value is shorter than the reference, the value is left-justified and blank-filled.

3.7.2 CHARACTER SUBSTRINGS

A character *substring* consists of one or more contiguous characters within a character variable or character array element (see 4.3 and 4.3.5). A substring name takes the following form:

cvname ([*first*]:[*last*])

cvname Name of a character variable, character array element, or character array section. See the restriction in the following paragraph.

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

Examples:

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

Substring notation cannot be applied to an unqualified array name; that is, a substring designator must follow either an array element name or an array section name. Examples:

CHAR(5) refers to array element 5.

CHAR(1:5) refers to elements 1 through 5.

CHAR(1:5)(1:10) refers to elements 1 through 5, positions 1-10.

3.7.3 ARGUMENTS OF TYPE CHARACTER

Actual arguments of type character can be character constants, the names of character variables or array elements, substrings, functions, or (in external procedures) character arrays. An actual argument associated with a character dummy argument must be of type character, with a length exceeding or equaling that of the dummy argument.

The preceding requirement for type agreement does not apply to character constants; as actual arguments these can be associated not only with character variables or arrays, but also, interpreted as Hollerith constants, with integer or real variables or arrays. See 5.2.3 and E.1.

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

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

If length *len* of a dummy argument of type character is less than the length of an associated actual argument, the leftmost *len* characters of the actual argument are associated with the dummy argument; that is, the rightmost characters are not part of the dummy argument.

When an actual argument is a character substring, the argument's length is the substring's length. Substring expressions in a substring name are evaluated immediately preceding argument association; the expression values remain constant as long as argument association continues.

If an actual argument is the concatenation of two or more operands, the argument's length is the sum of the operands' lengths. A character dummy argument whose length is specified as (*) must not appear as an operand for concatenation (see 5.2), except in a character assignment statement.

3.8 BOOLEAN TYPE (CFT77 EXTENSION)

A *Boolean* constant represents the bit pattern (sequence of 0's and 1's) of a single storage unit (64-bit Cray computer word). There are no Boolean variables, arrays, or array elements, and there is no Boolean type statement. A masking expression has a Boolean result, with each of its 64 bits representing the result of one or more logical operations on the corresponding bit of the expression's operands (see 5.5).

When an operand of a binary arithmetic or relational operator is Boolean, the operation is performed as if the Boolean operand had the same type as the other operand. If both operands are of type Boolean, the operation is performed as if they were of type integer. See tables 5-3 and 5-5.

No user-specified function can generate a Boolean result, but some intrinsic functions (non-ANSI) can generate Boolean results; see table B-8.

Following are some of the ways in which logical type differs from Boolean:

- Variables and arrays can be of logical type, and there is a LOGICAL type statement.
- A logical variable or constant represents only one value of true or false (rather than 64 separate bit values), and a logical expression yields one true or false value.
- Logical entities are invalid in arithmetic, relational, or masking expressions, while Boolean entities are valid. (Note, however, that results of relational expressions are logical type.)

A Boolean constant is written in one of three forms:

- The octal form contains 1 to 22 octal digits (0 through 7) followed by the letter B. 22 octal digits in a Boolean value correspond to the binary contents of a complete storage unit (64-bit word). In this case, the leftmost octal digit can be only 0 or 1, representing the content of the leftmost bit position (bit 0). Each successive octal digit specifies the contents of the next three bit positions until the last octal digit specifies the contents of the rightmost three bit positions (bits 61, 62, and 63). A Boolean value represented by fewer than 22 octal digits is right-justified; that is, it represents the rightmost bits of a Cray word: bits x through 63. Other bits are set to zero.
- The hexadecimal form contains the letter X followed by a string of 1 to 16 hexadecimal digits (0-9, A-F) enclosed by apostrophes or quotation marks. The hexadecimal digits may be preceded by an optional + or - sign; blanks are ignored. When a Boolean value contains 16 hexadecimal digits, their binary equivalents correspond to the content of every bit position in the storage unit (64-bit word). A Boolean value containing fewer than 16 hexadecimal digits is right-justified and zero-filled, as in the octal representation.

- A Hollerith constant is of type Boolean. When a character constant is used in a masking expression, the expression is evaluated as if the value were Hollerith, and a message is issued (see 5.2.3 and E.1). A Hollerith constant can have a maximum of 8 characters.

Examples:

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

The ANSI Fortran Standard does not provide for Boolean values.

3.9 POINTER TYPE (CFT77 EXTENSION)

A *pointer* is a variable whose value is used as the address of another entity, which is called a *pointee*. The POINTER type statement declares both the pointer and its pointee.

You can use pointers to access user-managed storage, by dynamically associating variables and arrays to particular locations in a block of storage. CFT77 pointers do not provide convenient manipulation of linked lists because, for optimization purposes, it is assumed that no two pointers have the same value. Pointers also allow the accessing of absolute memory locations.

A pointer's value occupies one storage unit (see 4.5.1). Its range of values depends on the size of memory for the machine in use.

Restrictions:

- A pointer cannot be pointed to by another pointer; that is, a pointer cannot also be a pointee.
- A pointee cannot appear in a SAVE, EQUIVALENCE, COMMON, or PARAMETER statement.
- A pointer cannot appear in a PARAMETER statement.
- A pointee cannot be a dummy argument; that is, it cannot appear in a FUNCTION, SUBROUTINE, or ENTRY statement.
- A function value cannot be a pointee.
- Integers can be converted to pointers; no other conversions involving pointers are allowed. An arithmetic expression containing a variable of type pointer has a result of type integer.
- A pointer variable cannot be declared to be of any other data type.
- A pointee cannot be of type character.

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

3.9.1 POINTER TYPE STATEMENT (CFT77 EXTENSION)

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

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

p Pointer to the corresponding *a*. *p* contains the word address of the location of *a*. Only a variable can be declared type pointer: constants, arrays, statement functions, and external functions cannot.

a Pointee of corresponding *p*; must be a variable name, array declarator, or array name. The value of *p* is used as the address for any reference to *a*; therefore *a* is not assigned storage. See restrictions in 3.9, preceding. If *a* is an array declarator, it can be constant, adjustable, or assumed-size.

Example:

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

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

An actual array that is named as a pointee in a POINTER statement is a *pointee array* (see 4.3.2). Its array declarator can appear in a separate type or DIMENSION statement or in the pointer list itself. In a subprogram, the dimension declarator in *a* can contain references to variables in a common block or to dummy arguments. As with adjustable array arguments to subprograms, each dimension's size is evaluated on entrance to the subprogram, not when the pointee is referenced. Example:

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

3.9.2 USING POINTERS

The pointer is a variable of type pointer and can appear in a COMMON list or be a dummy argument in a subprogram.

The pointee does not have an address until the pointer's value is defined: the pointee's value starts at the location specified by the pointer. Any change in the value of a pointer causes subsequent references to the corresponding pointee to refer to the new location.

Pointers can be assigned values in the following ways:

- A pointer can be set as an absolute address. Example:

  ```
  Q=0
  ```

- Pointers can have integer expressions added to or subtracted from them and may be assigned to or from integer variables. Example:

  ```
  P=Q+100
  ```

  However, pointers are not integers. For example, assigning a pointer to a real variable is not allowed (see table 5-3)

- The LOC function generates the address of a variable and can be used to define a pointer. Example:

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

The following example uses pointers in the ways just described:

```
SUBROUTINE SUB(N)
COMMON POOL (100000)
INTEGER JCB (128), WORD64
REAL A(1000),B(N),C(100000-N-1000)
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 above 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 associated with POOL. Similarly, WORD64 is the same as JCB(65), because JCB begins at address 0.

For purposes of optimization, the CFT77 compiler assumes that a pointee's storage is never overlaid on another variable's storage; that is, that a pointee is not associated with another variable or array. This kind of association occurs when a pointer has two pointees, or when two pointers are given the same value. Although these practices are sometimes used deliberately (such as for equivalencing arrays), results can differ depending on whether optimization is on or off. You are responsible for preventing such association. Example:

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

Because B and C have the same pointer, the assignment of 2.0 to C gives the same value to B; therefore B will print as 2.0 even though it was assigned 1.0.

As with a variable in common storage, a pointee, pointer, or argument to a LOC intrinsic function is stored in memory before a call to an external procedure and is read out of memory at its next reference. The variable is also stored before a RETURN or END statement of a subprogram.

DATA STRUCTURES, STORAGE, AND ASSOCIATION 4

This section describes the methods of using and storing data: constants, variables, arrays, and common blocks. It also describes how entities are associated. Relevant CFT77 statements are included.

4.1 CONSTANTS

A *constant* is an unchanging value. A *literal* constant is a constant represented by its value directly. A *symbolic* constant is a name representing a constant. Except within character constants, blank characters in a constant have no effect. A constant name is local to a program unit and cannot be used as the name of anything else except a common block.

A constant is given a symbolic name by the PARAMETER statement, in which the name's data type is established as if it were a variable. That is, the name can be declared in a preceding type statement, or the type can be implicit. If the symbolic name and the constant are of different types, the constant's type is converted to agree with the name, as in an assignment statement (see table 5-1). Example:

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

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

4.1.1 PARAMETER STATEMENT

A PARAMETER statement assigns a symbolic name to a constant. Because this value is established at compile time, it can be used in optimization.

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

p Symbolic name of a constant; must be integer, real, double precision, complex, or logical. Pointers are not allowed.

e Constant expression (see section 5 introduction); cannot include a function reference or array reference. In an exponentiation expression *x****y*, *y* must be an integer. Other requirements are referenced in the next paragraph.

If the data type of *p* is arithmetic, *e* must be an arithmetic constant expression (see 5.1). If *p* is of type character, *e* must be a character constant expression (see 5.2). If *p* is of type logical, *e* must be a logical constant expression (see 5.4). Any symbolic name in *e* must be a constant that has been defined in the same or an earlier PARAMETER statement.

Type conversion of a constant expression in a PARAMETER statement follows the same rules used for assignment statements (see table 5-1). The type of a symbolic name in a PARAMETER statement is specified by a previous type statement or IMPLICIT statement, or by default.

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

A symbolic name can be assigned a constant value only once in a program unit. Constants named in a PARAMETER statement can be referenced in any subsequent statement in the same program unit except in a format specification or to form a part of any other constant, such as either part of a complex constant.

Examples:

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

IMPLICIT LOGICAL(O)
PARAMETER (ON=.TRUE.,OFF=.FALSE.)

CHARACTER *(9) TITLE
PARAMETER (TITLE='CHAPTER 1')

CHARACTER * (*) STRING
PARAMETER (STRING='THIS IS TOO LONG TO COUNT')
```

```
COMPLEX Z
PARAMETER (Z = (1.0,2.0)/(7.0,8.0))

PARAMETER (FBASE=32, F=98.6, C=9./5.*(F-FBASE))
```

In the last example, FBASE is initialized first, then used later within the same PARAMETER statement in an expression to initialize C.

4.2 VARIABLES

A *variable* is a name whose value can be changed during program execution; it is unsubscripted and is not an array or array element. The term *subscripted variable* is used in some books to apply to an array element; this is not an ANSI term and is not used in this manual. *Scalar* and *simple* variables are merely variables as defined above, as distinguished from arrays or array elements. A variable name is local to a program unit (see 2.1.5).

Storage of a variable can be *static* or *stack*, as specified by the **alloc** or ALLOC option and the ALLOC directive. Storage allocation is discussed in subsection 4.5.2.

A variable has one data type throughout a program, as specified by the rules governing symbolic names (see 3.1). If a variable's type is implied by its name, the variable is not required to be declared before it appears in an executable statement, but it must have a value before being used in an expression. Definition of a variable value is discussed in 4.5.3.

A variable can have the same name as a common block. A variable or common block can have the same name as a dummy argument of a statement function, but the dummy argument name is local to the statement function definition statement. The data type (and, if type character, the length) is the same in all of these uses, except for use as a common block name.

A variable name has the following restrictions:

- It cannot appear in a PARAMETER, INTRINSIC, or EXTERNAL statement.
- It cannot be the name of an array, subroutine, main program, or block data subprogram; the entry name in an ENTRY statement; nor a NAMELIST group name.

4.3 ARRAYS

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

Example:

> One-dimensional array IFOUR has four elements. Its declarator in a type statement is INTEGER IFOUR(4). Its third element is IFOUR(3).

An array name with no subscript identifies the entire array in contexts that allow such use (see 4.3.9). In an EQUIVALENCE statement, an array name with no subscript identifies the first element of the array.

An *array element substring* is a character substring of a character array element (see 3.7.2).

An array is classified as dummy, pointee, or actual, depending on its relation to storage; and as constant-size, adjustable, or assumed-size depending on how its size is determined. Table 4-1 shows the possible combinations of these categories. An X indicates that the indicated pair is valid, such as actual and constant-size. The position of the word *Automatic* shows that this term applies to an array that is both adjustable and actual; this usage is a CFT77 extension.

Table 4-1. Possible Kinds of Arrays

	Actual	Dummy	Pointee
Constant-size	X	X	X
Adjustable	Automatic	X	X
Assumed-size		X	X

4.3.1 DUMMY, ACTUAL, AND POINTEE ARRAYS

An array is classified as dummy, pointee, or actual, depending on its use as a dummy argument, a pointee, or neither. These uses determine how an array is stored.

An *actual* array is allocated storage and is not a dummy or pointee array. It can appear in the main program or in procedure subprograms and can be constant-size or automatic (see 4.3.3).

A *dummy* array appears as a dummy argument in a procedure subprogram; it is associated with an actual array or array element through one or more procedure calls. A dummy array can be constant-size, adjustable, or assumed-size. See 4.3.8 concerning arrays as arguments.

Example:

```
PROGRAM                              SUBROUTINE SUBRTN (DUMARR)
REAL ACTARR(10)                      REAL DUMARR(10)
...                                  ...
CALL SUBRTN (ACTARR)
...
```

Actual array ACTARR above is used as an argument in the call to subroutine SUBRTN. The subroutine declares dummy array DUMARR, which becomes associated with ACTARR. Storage for ACTARR is allocated by the calling program unit; this same storage is accessed by references to DUMARR during the call to SUBRTN.

A *pointee* array is the object of a pointer (see 3.9); it can be constant, adjustable, or assumed-size but cannot be a dummy argument. It is not allocated storage; rather, its storage begins at the pointer address. The name of a pointee array must appear in a POINTER statement, either alone or as part of the array declarator; the declarator can also appear in a DIMENSION or type statement. Subsection 3.9.2 includes an example in which a pointee array is stored in a common block.

The ANSI Fortran Standard does not provide for pointers or pointee arrays.

4.3.2 CONSTANT, ADJUSTABLE, AND ASSUMED-SIZE ARRAYS

An array is constant-size, adjustable, or assumed-size, depending on how its size is determined. Each category has a corresponding kind of declarator (see 4.3.5).

A *constant-size* array has dimensions that do not vary in size; that is, the dimension bounds in the array declarator are arithmetic constant expressions (see 5.1).

An *adjustable* array is a dummy, pointee, or automatic array whose size is determined during program execution, as specified by the array declarator. Each reference to a subprogram can specify dimension sizes for an adjustable array in the subprogram; the sizes are constant during subprogram execution. Variables defining adjustable dimension bound expressions (see 4.3.5) can be redefined or become undefined during execution of the subprogram, with no effect on the bound or on array size.

An *assumed-size* array is a dummy or pointee array whose last dimension is of an unknown size (specified as "*") that is assumed to be large enough for all references made to the array. An assumed-size array cannot be used in an I/O statement as an item in an I/O list, as a format identifier, or as a unit identifier for an internal file. An assumed-size array name cannot be used as a whole array reference (see 4.3.11) but can be used to form an array section (see 4.3.10).

The ANSI Fortran Standard does not provide for array syntax or array sections.

4.3.3 AUTOMATIC ARRAYS (CFT77 EXTENSION)

An *automatic* array is an actual array, appearing only in a procedure subprogram, whose size is determined at runtime in the same way as that of an adjustable array. An automatic array is typically needed for scratch storage within a subprogram. Storage for the array is allocated when the procedure is entered and released on exit. As with an adjustable array dummy argument, an automatic array declaration cannot be changed once it is evaluated.

The ANSI Fortran Standard does not provide for automatic arrays.

4.3.4 DIMENSION STATEMENT

The DIMENSION statement specifies the symbolic names and dimension specifications of arrays, by means of array declarators. Array declarators can also appear in COMMON statements, type statements, and POINTER statements.

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

a(d) Array declarator; see 4.3.5, following.

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 in an array declarator in a program unit.

4.3.5 ARRAY DECLARATORS

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

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

a Symbolic name of the array

$[l_n:]u_n$ Dimension declarator, one for each dimension in the array. *u* and *l* are integer expressions called the *dimension bound expressions*. Dimension declarators and dimension bound expressions are discussed in the following text.

- *l* Lower bound of the dimension; default is 1.
- *u* Upper bound of the dimension. This is specified as * for the last dimension of an assumed-size array.

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

A *dimension bound expression* specifies the upper or lower bound of a dimension in an array declarator. The expression must be scalar and must

have an integer result. Any dimension bound expression can contain constants and symbolic names of constants. Adjustable and assumed-size dimension bound expressions can also contain: functions; array elements; or variables of any type, provided the expression's result is an integer.

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

Examples:

```
DIMENSION ARRAY1 (34,0:24,1:34), ARRAY2 (64), Z7144X (5:10,-2:20)

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

INTEGER TABLE (3,IVAL,IRUNS,2,2), TAB(6:IVALX,MAT:10)
```

In the first example, arrays ARRAY1 and ARRAY2 can be implicitly real or can be declared another data type in a separate statement. In the last two examples, the use of variables defines adjustable dimensions; these are used only in procedure subprograms.

4.3.5.1 Kinds of array declarators

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

Actual, dummy, pointee - An *actual* array declarator declares an actual array and must be constant-size or automatic. Actual array declarators are permitted in DIMENSION, COMMON, and type statements, with the exception that automatic declarators are not permitted in COMMON statements.

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

A *pointee* array declarator has the same requirements as a dummy array declarator, except that it can appear in a POINTER statement. If it does not appear in a POINTER statement, the array name must appear in a POINTER statement, and the declarator must appear in a type or DIMENSION statement.

Constant, adjustable, assumed-size, automatic - In a *constant* array declarator, the dimension bound expressions contain only constants and names of constants; that is, they contain no variables, functions, or array elements.

Adjustable and *automatic* array declarators appear only in procedure subprograms and have identical properties, but apply to adjustable and automatic arrays, respectively. In either kind of declarator, at least one of the dimension declarators contains one or more variables, functions, or array elements. In a subprogram containing an automatic or adjustable array reference, each variable or array element in the array's declarator must be named as follows:

- Automatic (actual) array declarator: in every dummy argument list or in a COMMON statement

- Adjustable (dummy or pointee) array declarator: in every argument list that contains the array name, or in a COMMON statement

Array elements used in an adjustable or automatic array declarator must be of arrays previously dimensioned.

In an *assumed-size* array declarator, the upper bound of the last dimension declarator is an asterisk, indicating a value that is not specified but is assumed to be large enough for any reference made to the array. The other dimensions can be either constant or adjustable.

4.3.6 ARRAY ELEMENTS AND SUBSCRIPTS

The *subscript* of an array element name identifies the element's position in the array, and consists of a group of subscript expressions separated by commas and between parentheses. Within a subscript, each subscript expression specifies the element's position in one array dimension. The *subscript value* is the ordinal number of an element's position in the array's storage sequence; see 4.3.6.1.

The format of an array element name is as follows:

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

a Array name

s Subscript expression; must be scalar and integer type.

Examples:

```
A(1)            TABLE(45,27,106)          TIME(NINT(B+4.5/PI))
```

NOTE

In ANSI usage, the term *subscript* refers to a group of subscript expressions that completely specify an element's position within an array. In common usage the term often refers to just one subscript expression within a group.

4.3.6.1 Array subscripts and storage sequence

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

Although an array is arranged in dimensions for programming purposes, it is stored in a single ordinal sequence. Each array element's position in the sequence is specified by its subscript; this consists of one or more subscript expressions, each of which specifies the element's position within one array dimension. In relation to the storage sequence, the leftmost expression is incremented most frequently, and each expression to the right is incremented less frequently.

If an ordered pair of numbers represents an element's row and column, respectively, the storage sequence for a two-dimension array corresponds to the elements going down each column in succession. Figure 4-2 shows the storage sequences for the arrays shown in figure 4-1.

Table 4-2 illustrates the conversion of subscript values to ordinal positions within an array's storage sequence. CFT77 uses A registers for subscript calculations. Overflow on intermediate values is not detected; very large values in subscript expressions can produce unpredictable results. Register sizes and values are as follows:

Computer	Register Size	Maximum Intermediate Value
CRAY-1 or CRAY X-MP computer systems	24-bit	$2^{23}-1$ (or 8,388,607)
CRAY-2 computer system	32-bit	$2^{31}-1$ (or 2,147,483,647)

Notice that the above maximum values are only for intermediate values in subscript calculations, and are distinct from the maximum number of words in an array; see 4.3.7, following.

Table 4-2. Subscript Evaluation

As Shown in ANSI Fortran Standard	As Computed by CFT77 Compiler
1 dimension: declarator $(l_1:u_1)$; subscript (s_1)	
$1 + (s_1 - l_1)$	$1 + s_1 - l_1$
2 dimensions: declarator $(l_1:u_1, l_2:u_2)$; subscript (s_1, s_2)	
$1 + (s_1 - l_1)$	$1 + s_1$
$+ (s_2 - l_2)*d_1$	$+ s_2*d_1$
	$- (l_1 + l_2*d_1)$
3 dimensions: declarator $(l_1:u_1, l_2:u_2, l_3:u_3)$; subscript (s_1, s_2, s_3)	
$1 + (s_1 - l_1)$	$1 + s_1$
$+ (s_2 - l_2)*d_1$	$+ s_2*d_1$
$+ (s_3 - l_3)*d_2*d_1$	$+ s_3*(d_2*d_1)$
	$- (l_1 + l_2*d_1 + l_3*(d_2*d_1))$
n dimensions: declarator $(l_1:u_1, \ldots l_n:u_n)$; subscript $(s_1, \ldots s_n)$	
$1 + (s_1 - l_1)$	$1 + s_1$
$+ (s_2 - l_2)*d_1$	$+ s_2*d_1$
$+ (s_3 - l_3)*d_2*d_1$	$+ s_3*(d_2*d_1)$
$+ \ldots$	$+ \ldots$
$+ (s_n - l_n)*d_{n-1}*d_{n-2}*\ldots*d_1$	$+ s_n*(d_{n-1}*\ldots*d_1)$
	$- (l_1 + l_2*d_1 + l_3*(d_2*d_1)$
	$+ \ldots + l_n*(d_{n-1}*\ldots*d_1))$

l = Lower bound of dimension declarator
u = Upper bound of dimension declarator
s_i = Subscript expression $(l_i \leq s_i \leq u_i)$
d_i = Dimension size $(u_i - l_i)+1$

The number of subscript expressions normally equals the number of dimension declarators in the array declarator. When fewer subscript expressions are used, these are used for the leftmost dimensions, and the lower bound is assumed for the missing expressions; a warning message is issued. For example, in a three-dimensional array A with lower dimension bounds of 1, A(7) is interpreted as A(7,1,1).

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

4.3.7 ARRAY SIZE

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

CFT77 allows a maximum array size of 4,194,304 Cray computer words (CRAY-1 and CRAY X-MP computer systems); 16,777,215 words (CRAY X-MP computer system with extended memory addressing); or 268,435,456 words (CRAY-2 computer system). The size is further restricted by the Cray computer system in use, the executable program size, and the amount of memory required, beyond that for the executable program and related data. These maximum sizes are independent of the maximum values allowed in calculating subscript values (see 4.3.6.1).

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

4.3.8 ARRAYS AS ARGUMENTS

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

Actual and dummy array elements are associated by their subscript values. When the actual argument is an array element of subscript *s*, the element's address is used as the dummy array's base address; element *t* in the dummy array is then associated with element *s*+*t*-1 in the actual array.

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

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

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

4.3.9 USE OF ARRAY NAMES

The use of an array name with no subscript, or an array section name (see 4.3.10) implies that the number of values to be processed equals the number of elements in the array or array section.

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

- Array expression
- Array declarator
- List of arguments
- COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statement
- Type statement
- POINTER statement, as a pointee
- Input/output statement: as a format identifier, in a list, or as a unit identifier for an internal file. These do not apply to an assumed-size array or array expression.

One Dimension	
1	ARX(1)
2	ARX(2)
3	ARX(3)
4	ARX(4)
5	ARX(5)
6	ARX(6)

Two Dimensions

	1	2	3	4
1	1,1	1,2	1,3	1,4
2	2,1	2,2	2,3	2,4
3	3,1	3,2	3,3	3,4
4	4,1	4,2	4,3	4,4
5	5,1	5,2	5,3	5,4
6	6,1	6,2	6,3	6,4
7	7,1	7,2	7,3	7,4
8	8,1	8,2	8,3	8,4
9	9,1	9,2	9,3	9,4

Three Dimensions

3

	1	2
1	1,1,3	1,2,3
2	2,1,3	2,2,3

2

	1	2
1	1,1,2	1,2,2
2	2,1,2	2,2,2

1

	1	2
1	1,1,1	1,2,1
2	2,1,1	2,2,1
3	3,1,1	3,2,1
4	4,1,1	4,2,1
5	5,1,1	5,2,1
6	6,1,1	6,2,1
7	7,1,1	7,2,1

Declaration statements:

```
DIMENSION ARX(6)
REAL ARY(9,4)
INTEGER ARZ(7,2,3)
```

Array declarator:	ARX(6)	ARY(9,4)	ARZ(7,2,3)
Data type:	Real	Real	Integer
Dimension sizes:	6 elements	9 and 4 elements	7, 2, and 3 elements
Total elements:	6	36	42
Number of words:	6	36	42

Figure 4-1. Array Specification and Size

1	ARX(1)
2	ARX(2)
3	ARX(3)
4	ARX(4)
5	ARX(5)
6	ARX(6)

1	ARY(1,1)
2	ARY(2,1)
3	ARY(3,1)
4	ARY(4,1)
5	ARY(5,1)
6	ARY(6,1)
7	ARY(7,1)
8	ARY(8,1)
9	ARY(9,1)
10	ARY(1,2)
11	ARY(2,2)
12	ARY(3,2)
13	ARY(4,2)
14	ARY(5,2)
15	ARY(6,2)
16	ARY(7,2)
17	ARY(8,2)
18	ARY(9,2)
19	ARY(1,3)
20	ARY(2,3)
21	ARY(3,3)
22	ARY(4,3)
23	ARY(5,3)
24	ARY(6,3)
25	ARY(7,3)
26	ARY(8,3)
27	ARY(9,3)
28	ARY(1,4)
29	ARY(2,4)
30	ARY(3,4)
31	ARY(4,4)
32	ARY(5,4)
33	ARY(6,4)
34	ARY(7,4)
35	ARY(8,4)
36	ARY(9,4)

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

Notice that the leftmost subscript expression is incremented most frequently. Each expression to the right is incremented less frequently.

Figure 4-2. Storage Sequence for Arrays in Figure 4-1

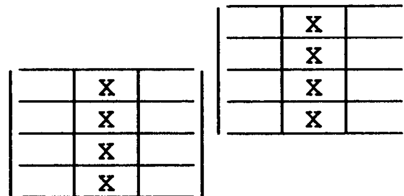
4.3.10 ARRAY SECTION (CFT77 EXTENSION)

An *array section* is a group of elements, within an array, that can be used as an operand in an array expression (see 4.3.11). An array section name has the same form as an array element name, except that one or more subscript expressions are replaced by *section selectors,* each of which specifies an entire dimension or part of a dimension. The simplest form of a section selector is a colon, which specifies a whole dimension.

Example:

```
Array declarator:     A(4,3,2)
Array section name:   A(:,2,:)
```

The above section name specifies an array section with two dimensions. The section's first dimension corresponds with A's first dimension, and the second with A's third dimension. All elements of the resulting array section occupy the second column of A. Elements of A that are part of the array section are indicated by X's in the following diagram:



In an array section, a dimension's size equals the number of elements selected by the section selector for that dimension. The size of an array section equals the product of its dimension sizes.

4.3.10.1 Uses and restrictions

The following points apply to the use of array sections:

- Array sections can be formed from character arrays (see 3.7.2).
- Array sections are permitted only in assignment statements, on either side of the equal sign.
- Array sections may not be used as actual arguments to external functions or statement functions, but they may be actual arguments to intrinsic functions.
- A section of an assumed-size array is allowed only if the section selector for the last dimension is a subscript expression, an indexed section selector with the upper bound specified, or a vector-valued section selector with a known size.

4.3.10.2 Array section name

The format of an array section name is as follows:

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

a Array name

s Section subscript expression, corresponding to one dimension in an array section. The expression can be a subscript expression, behaving the same as a subscript expression in an array element name, or a section selector. A section selector can be either indexed or vector-valued; see the following text.

4.3.10.3 Indexed section selectors

An *indexed* section selector is a section subscript expression that selects the elements of one dimension within a certain range and at a certain interval. It takes the following form:

```
[l]:[u][:i]
```

l, u Subscript expressions selecting (respectively) the lower and upper bounds for one dimension of an array section. *l* and *u* must be integer scalar expressions.

l and *u* default to the lower and upper bounds for the dimension in the array that contains the array section. *l* and *u* must be greater than or equal to the array dimension's lower bound and less than or equal to the dimension's upper bound. The upper bound *u* must be specified for the last dimension of a section of an assumed-size array.

i Increment, cannot be 0; default is 1.

A specifier consisting of only a colon uses all defaults and therefore specifies the entire dimension. A specifier of 4:10:2 specifies positions 4, 6, 8, and 10 of a dimension.

Notation for substrings of character array sections is discussed in 3.7.2.

An indexed section selector selects the elements from *l* to *u* in increments of *i*. Thus the elements selected are *l*, *l*+*i*, *l*+2*i*, and so on. The total number of elements identified, which must be positive and greater than zero, is given by the following expression:

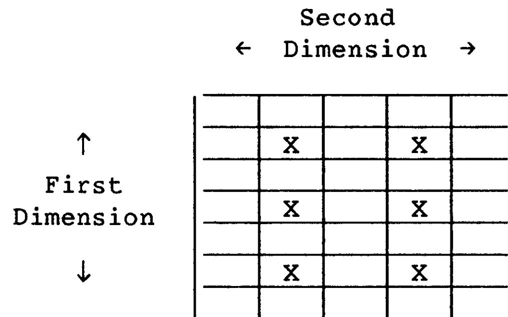
```
INT((u-l+i)/i)
```

Examples:

(1)

```
REAL D (7,5)
.... = D(2:6:2,2:4:2)
```

The array section name in the second line identifies rows 2 through 6 in increments of 2 (first dimension), and columns 2 through 4 in increments of 2 (second dimension). The following diagram shows the section within the array.



(2)

```
REAL A(10)
... = A(1:10:2)
```

The array section name in the second line specifies the following elements:

4.3.10.4 Vector-valued section selectors

A *vector-valued* section selector is the name of a one-dimensional array (not assumed-size), array expression, or array section; it must be type integer. Vector-valued section selectors allow an array section to include any arbitrary group of array elements.

A vector-valued section selector selects elements in positions given by the elements of the section selector. The elements of the section selector must be greater than or equal to the lower bound of the dimension, and less than or equal to the upper bound. The number of elements selected is equal to the number of elements in the vector-valued section selector.

Example:

```
INTEGER A(10), I(20,20), S1(5), S2(5)
```

The following are array section names using vector-valued section selectors:

A(S1)

The array section's elements are selected by array S1.

A(S1+S2)

S1+S2 is an array expression; the resulting array is a vector-valued selector.

A(I(S1,3))

Array section I(S1,3) is a one-dimensional array section selected by array S1 from the third column of array I. This array section is in turn a vector-valued section selector for an array section drawn from array A.

A problem occurs when two elements of a vector-valued section selector have the same value. When this occurs, two elements of the specified array section represent the same element of the original array. When an assignment includes such an array section, the order of operations is not guaranteed, leaving some values unpredictable. CFT77 implements array sections as specified in the ANSI Fortran 8X standard, which explicitly allows a reversed order of assignment.

Example:

```
      INTEGER A(5),B(5),C(5),D(5),IX(5)
      DATA IX/1,2,3,4,4/                         ! Repeated value
      DO 10, J=1,5
        A(J)=J**2
   10 CONTINUE
      B(IX)=A                       ! Assignment order not guaranteed
      C=B(IX)
      D=A(IX)
```

Because IX(4) and IX(5) have the same value, the above array sections cause one element of array A or B to be, in effect, specified twice in the same assignment statement: that is, A(IX(4)) and A(IX(5)) are both element A(4). When the above code was run, the final values of arrays A, B, C, and D were as shown below; but this result could change, depending on optimization and the particular release of CFT77.

A:	1	4	9	16	25
B:	1	4	9	25	0
C:	1	4	9	25	25
D:	1	4	9	16	16

4.3.11 ARRAY EXPRESSIONS (CFT77 EXTENSION)

An *array expression* is an expression (arithmetic, relational, character, or logical) in which one or more primaries are array operands. An *array operand* is an operand that is an array reference with no subscript (a *whole array reference*), an array section reference, or an array expression. The name of an assumed-size array cannot be an array operand.

The result of an array expression is an array value. An array expression can be used only in an assignment statement.

The ANSI Fortran Standard does not provide for array expressions.

4.3.11.1 Conformance of array operands

When an operator (arithmetic, relational, character, or logical) operates on a pair of operands, of which at least one is an array operand, the operands must be *conformable*. An array or array section is conformable with either of the following:

- An array or array section of the same *shape;* that is, one with the same number of dimensions and the same number of elements in corresponding dimensions. Assumed-size arrays are not conformable, but a section of an assumed-size array can be conformable.

- A scalar item

Conformance is checked for all arrays and array sections whose sizes are known at compile time. If **-e o** is on the **cft77** command or ON=O is on the CFT77 control statement, conformance is checked at runtime for arrays and array sections whose sizes cannot be determined at compile time.

Example:

```
DIMENSION A(5), B(10,5), C(5:14,6:2)
```

Conformable:

A	with	B(1,:)	
A	with	C(6:10,2)	
B	with	C	Same shaped arrays: (14-5)+1=10 and (6-2)+1=5

Not conformable:

A	with	B	(unequal number of dimensions)
B(:,2)	with	C(1:5,2)	(unequal number of elements in one dimension)
B(:,1)	with	C	(unequal number of dimensions)

4.3.11.2 Order of operations

An array operation is performed element-by-element on corresponding elements of the array operands, the correspondence established by their subscripts. The operation's result has the same shape as the array operand(s). Each element of the array result has the value of the indicated operation performed on corresponding elements of the array operands. The order in which the element-by-element operations are performed is not specified.

Example:

```
      DIMENSION A(10),B(10),C(10),D(10)
      A=(B+C)*D
```

For the array operation shown in the second statement above, elements B(2) and C(2) could be added before B(1) and C(1), and the result of (B(2)+C(2)) could by multiplied by D(2) before B(1) and C(1) are added.

In an array syntax assignment statement, the expression's result is assigned as if the entire right-side expression and left-side subscript were evaluated first. That is, the destination name may be referenced in the right-side expression or left-side subscript. Examples:

(1)

```
      REAL X(4)                DATA X/1.,2.,3.,4./
      X(2:4) = X(1:3)
```

The last statement above yields X(1)=X(2)=1.0; X(3)=2.0; and X(4)=3.0. However, the following DO-loop does not have the same result, simply assigning 1.0 to elements X(2), X(3), and X(4).

```
      DO 10 I=2,4               !Unintended result
        X(I) = X(I-1)
   10 CONTINUE
```

(2)

```
      A(2:11) = A(1:10)
```

The above statement is equivalent to

```
      TEMP(1:10) = A(1:10)   A(2:11) = TEMP(1:10)
```

but not to

```
      DO 20 I=1,10              !Unintended result
        A(I+1) = A(I)
  20  CONTINUE
```

(3)

```
      INTEGER I(5)             DATA I/1,2,3,4,5/
      I(I(5:1:-1)) = I
```

This assignment requires pre-evaluating of both array I and the left-side subscript. The creation of temporary arrays used in the pre-evaluation affects runtime performance.

Examples of array expressions and assignment statements:

```
      DIMENSION A(10), B(10,10), C(10,10,20)

(1)   A = B(:,3)

(2)   C(:,I,1:19:2) = B * B

(3)   A = B(3*I,10:1:-1) / C(:,5,5)
```

In this example, B(3*I,10) is divided by C(1,5,5); B(3*I,9) by C(2,5,5), and so on.

```
(4)   A(:) = A(10:1:-1)
```

The above statement reverses the elements in one-dimensional array A.

```
(5)   A=5
```

The above statement assigns 5 to every element of array A.

4.3.11.3 Array operands in intrinsic functions

Array operands can be used as arguments to intrinsic functions that take arguments; the result is an array that can itself be used as an array operand. Functions with scalar arguments can be used in array expressions in the same way that other scalar operands are used.

An intrinsic function that takes more than one argument, such as MAX, operates on array operands element by element.

Examples:

(1)

```
      REAL A(5,5),B(5,5)
      A = SIN(B)
```

(2)

```
      REAL S(3,3), T(3,3), U
      S = SQRT(T) + U
```

Each element of array S is assigned the sum of scalar variable U and the square root of the corresponding element of array T.

(3)

```
      REAL U(4,4,4),V(4,4,4),W
      U(:,:,3) = LOG(V(2,:,:)) + W
```

(4)

```
      INTEGER I(10,12),J(10,12),K(10,12),L
      I = MAX(J,K,L)
```

Each element of array I is assigned the highest among the corresponding elements in arrays J and K, and scalar variable L.

(5)

```
      INTEGER L(5,10),M(5,10),N(5,10),I(5,10),J
      L = CVMGP(M,N,I)
      L = CVMGM(M,N,J)
```

The CVMG intrinsic functions operate on array operands in the same way as do other multi-argument intrinsic functions.

4.4 DATA STATEMENT

The DATA statement provides initial values for variables, arrays, and array elements. A DATA statement can be intermixed with specification statements but must follow type and DIMENSION statements for variables appearing in the DATA statement. Outmoded features of the DATA statement are described in E.6.

Entities appearing in DATA statements are treated as if they had been named in a SAVE statement, even when stack mode is specified.

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

The ANSI Fortran Standard does not specify storage allocation methods.

The format of a DATA statement is as follows:

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

nlist List of variable names, array names, array element names, substring names, and implied-DO lists separated by commas. Subscript and substring expressions must be integer constant expressions (see 5.1); subscript expressions can include implied-DO variables. Declaration statements affecting the names in *nlist* must precede the DATA statement. *nlist* cannot include names of constants, dummy arguments, functions, entities in blank common, or entities associated with entities in blank common.

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

- *c* Constant or the symbolic name of a constant
- *r* Nonzero, unsigned, integer constant or the symbolic name of such a constant

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

Examples:

```
DATA PI/3.14/, E/2.73/, G/4.77/
DATA C(1),C(2),C(3),C(4),C(5)/1.0,2.0,3*0.0/ C(10)/1.0/
```

Initial values of the entities are defined by the correspondence between *clist* and *nlist* elements. Counting as separate items the elements of any arrays in *nlist*, and counting an *r*c* entry in *clist* as *r* items, the *i*th item in *nlist* becomes defined with the *i*th value from *clist*. Counted in this way, the same number of items must be specified by an *nlist* and its corresponding *clist*. (See E.6 for an exception to this rule.)

Examples:

(1)

```
REAL A(5)
DATA A/5*0.0/
```

Array A above has five elements, and the DATA statement assigns zero to each element.

(2)

```
INTEGER J(2,3)
DATA J/1,2,3,4,5,6/
```

Array J above has six elements, which are assigned different values in the *clist* of the DATA statement. That is, J(1,1)=1, J(2,1)=2, J(1,2)=3, J(2,2)=4, J(1,3)=5, and J(2,3)=6.

(3)

```
DIMENSION GRID(2,3), KBUF(10,200,2)
PARAMETER (XCON=6.0)
DATA GRID/11.0,21.0,12.0,22.0,13.0,23.0/, KBUF/4000*XCON/
DATA I/1/ K/2000/
PARAMETER (NEG=-6)
INTEGER NB(10)
DATA NB/-3,7*-4,2*NEG/
```

The above code shows the interaction of several specification statements with DATA statements. Notice that NEG is defined in a PARAMETER statement and used in the last DATA statement.

4.4.1 IMPLIED-DO LIST IN A DATA STATEMENT

An implied-DO list allows a DATA statement to initialize a group of values systematically as in a DO-loop. The implied-DO is used frequently for initializing arrays. The following format represents one item in an *nlist* as shown in the DATA statement format; the values assigned are contained in a subsequent *clist*.

(*dlist*, *dvar* = *init*,*lim*[,*incr*])

dlist — List of array element names and implied-DO lists separated by commas. *dlist* cannot contain substring names, even if they are substrings of array elements.

dvar — Name of an integer variable called the *implied-DO variable*. *dvar* is initially defined with *init* and is incremented by *incr* on each iteration of the loop. The range of this variable is *dlist*; that is, it does not conflict with other uses of the same name. *dvar* must be used in the subscript list of every array referenced in *dlist*.

init, *lim*, and *incr*

Initial value, limit, and increment of the implied-DO variable; must be integer constant expressions (see 5.1), which can contain references to *dvar*. The expressions can contain implied-DO variables of other implied-DO lists containing this implied-DO list within their ranges. *incr* cannot be zero, and defaults to 1.

The range of an implied-DO list is the list *dlist*. The trip count and values of the implied-DO variable *dvar* are established as for a DO-loop except that the trip count must be positive. Interpretation of an implied-DO list in a DATA statement causes each item in the list *dlist* to be specified once for each iteration, so that appropriate values are substituted where implied-DO variables are referenced.

Examples:

(1)

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

The first ten values of array A above are set to 1.

(2)

```
      INTEGER J(100)
      DATA (J(I),J(I-1),I=5,100,5)/10*0,10*1,10*0,10*1/
```

The implied-DO in the above DATA statement selects 40 elements of array J and assigns values of 0 and 1 to them in groups of ten each.

(3)

```
      DIMENSION A(5), B(5)
      DATA ((A(I),B(I)),I=1,5)/10*0.0/     ! Incorrect format
```

The implied-DO list in the above DATA statement is incorrect. The list of array element names (A(I),B(I)) is treated as a single complex variable.

4.4.2 DATA TYPES IN A DATA STATEMENT

When the *nlist* entity is of type integer, real, or double-precision, the corresponding *clist* constant is converted, if necessary, to the type of the *nlist* entity according to the rules for arithmetic conversion, as shown in table 5-3.

The type of the *nlist* entity and that of the *clist* constant must agree when either is of type character or logical. (An exception for type character is described in E.6).

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

4.4.3 ENTITIES THAT CAN APPEAR IN A DATA STATEMENT

Any variable or array element can be initially defined in a DATA statement except for the following:

- An entity that is a dummy argument
- A variable in a function subprogram whose name is also the name of the function subprogram
- A pointee
- An automatic array or an element of an automatic array
- An entity in blank common or task common

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

4.5 STORAGE AND ASSOCIATION

This subsection discusses the storage of variables, arrays, and common blocks, and the way entities become associated.

4.5.1 STORAGE UNITS AND SEQUENCES

A *numeric storage unit* is a Cray word of 64 bits; a *character storage unit* is an 8-bit byte. A *storage sequence* is a contiguous group of storage units with a consecutive series of addresses. Each array and each common block is stored in a storage sequence. The size of a storage sequence is the number of storage units it contains. Two storage sequences are *associated* if they share at least one storage unit.

An integer, real, or logical value occupies one numeric storage unit. A character value is represented as an 8-bit ASCII code, packed eight characters per word; the storage size depends on the value's length specification. A double-precision or complex value uses a storage sequence of two numeric storage units, with the first storage unit containing the most significant bits of a double-precision value or the real part of a complex value, and the second storage unit containing the least significant bits of a double-precision value or the imaginary part of a complex value.

The ANSI Fortran Standard does not specify the relationship between storage units and computer words, nor does it specify any relation between numeric and character storage units.

4.5.2 STATIC AND STACK STORAGE

With *static* storage, any variable that is allocated memory occupies the same address throughout program execution. Allocation is determined before program execution. Code using static storage is not reentrant and does not adapt to multitasking.

A *stack* is an area of memory where storage for variables is allocated when a subprogram or procedure begins execution and is released when execution completes. The stack expands and contracts as procedures are entered and exited. The amount of memory available for the stack is determined by the STACK directive available with SEGLDR; see the Segment Loader (SEGLDR) Reference Manual, publication SR-0066.

Variables are allocated to storage according to the following criteria:

- User variables appearing in COMMON statements are always allocated in the order they appear in the source program.
- User variables that are defined or referenced in a program unit, and that also appear in SAVE or DATA statements, are allocated to static storage, but not necessarily in the order shown in your source program.
- Other referenced user variables are assigned to the stack if **-a stack** is on the **cft77** command line, ALLOC=STACK is on the CFT77 control statement, or CDIR$ ALLOC=STACK appears in the program. If you have not specified stack storage, referenced variables are allocated to static storage. This allocation does not necessarily depend on the order in which the variables appear in your source program.

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

- In either allocation mode, heap allocation is used for TASK COMMON variable and some compiler-generated temporary data such as automatic arrays and array temporaries. A *heap* is memory that is dynamically by the system as a program runs.

NOTE

Unreferenced user variables not appearing in COMMON statements are not allocated.

4.5.3 DEFINITION

A *defined* variable or array element has a value. An *undefined* variable or array element does not have a predictable value. Once defined, a variable or array element keeps a value until it becomes undefined or redefined.

All variables and array elements are initially undefined and can be defined before or during program execution. They can be defined in the following ways:

- By an assignment statement (see 5.1.1, 5.2.1, and 5.4.1)

- By an input statement (see section 7)

- By a DATA statement (see 4.4)

- Through association with other entities (see 4.5.5)

- When used as a dummy argument in a subprogram, when the subprogram is called and the corresponding actual argument is defined (see 2.5)

- By an ASSIGN statement (only for a variable representing a statement label); see 6.4.4.

In addition, DO variables, implied-DO variables, and specifiers in I/O statements become defined when these statements and constructs are used.

An *initially defined* variable or array element is assigned a value in a DATA statement. Constants are always defined and are never redefined. A function's value is defined only when it is evaluated.

When an entity of a given type becomes defined, all totally associated entities of different types become undefined. When an entity not of character type becomes defined, all partially associated entities become undefined. However, if two partially associated entities are of types real and complex, one entity can become defined without causing the other to become undefined.

If one array element is redefined, the whole array is considered redefined. This can affect optimization.

4.5.4 SAVE STATEMENT

A SAVE statement retains the definition status of specified entities 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.

When a subprogram is called recursively (directly or indirectly), inputs to the recursive invocation should be passed as arguments or within a common block. Any SAVE statement has not yet taken effect, so you should not assume that a variable's current value will be in storage when the recursive invocation begins. That is, input variables used in recursive calls should be passed as in any other subprogram call.

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

a Variable name; array name; or common block name preceded and followed by a slash. If *a* is omitted, all common blocks, variables, and arrays that can legally appear in a SAVE statement are assumed specified. A name must not appear more than once in the SAVE statements of a program unit.

The names of dummy arguments, pointees, automatic arrays, or function result variables must not be specified in a SAVE statement. Variables and arrays within a common block can be specified only if the entire block is included.

The ANSI Fortran standard specifies that a common block that is named in a SAVE statement must also be specified in every subprogram where the common block appears. This is not enforced by CFT77.

A SAVE statement in the main program has no effect.

All referenced entities specified in a SAVE statement are assigned to static storage (see 4.5.2), with the following exception: when optimization is enabled, local variables or arrays that are not referenced or used might not be assigned storage, and in such cases are unaffected by the SAVE statement.

The ANSI Fortran Standard does not specify storage allocation methods.

4.5.5 ASSOCIATION OF ENTITIES

Two entities are associated if their names represent the same storage location, either within a program unit or in different program units: that is, entities are associated if they use the same storage, or if their storage sequences overlap. Character entities must not be associated with arithmetic or logical entities.

Association of entities across program units allows a subprogram to process data as needed by another program unit. This is accomplished by the use of either arguments or common blocks. Arguments allow the calling program unit to specify, in effect, an address to be accessed; this associates entities in the called and calling program units (see 2.5). With a common block, each program unit independently declares a group of entities that occupy a common area of storage; corresponding entities declared by different program units are then associated (see 4.6). Data that can be referenced by more than one program unit is called *global* in Cray parlance; see 4.5.5.2.

Within a program unit, an EQUIVALENCE statement associates entities. Such association is used to organize storage in large common blocks (see 4.6.3).

Totally associated entities have the same storage sequence. *Partially associated* entities share part but not all of a storage sequence. All entities using a given storage unit are affected by the unit's value or undefined status; totally associated entities of the same type have the same values and definition status.

Partial association can exist between a double-precision or complex value and a second value of type integer, real, logical, double-precision, or complex; or between two character entities. Partial association can occur through the use of COMMON, EQUIVALENCE, or ENTRY statements. (In a function subprogram, all entry names are associated with one another and with the function name.) Partial association must not occur through argument association except for character arguments.

Partial association is illustrated in 4.5.6.

****

CAUTION

The order in which CFT77 stores variables may differ from that used by other compilers, such as CFT. Programs written for these compilers give invalid results when compiled with CFT77 if they use *weak implicit association,* described in the following subsection.

****

4.5.5.1 Implicit association

With many Fortran compilers, including CFT, entities can be associated based on assumptions about how storage is allocated. For example:

```
INTEGER T(5)
INTEGER A,B,C,D,E
EQUIVALENCE (T(1),A)
```

With such compilers, you can assume that variables A through E are stored in the order declared in the above INTEGER statement; the EQUIVALENCE statement would then associate element T(2) with B, and so on. This is *weak implicit association* because it is merely a by-product of certain compiler implementations.

Under CFT77, however, you should not make assumptions about storage sequences except for those of an array or of the entities declared in a COMMON statement. Only in these cases can association be reliably based on implied positions, such as when two arrays are equivalenced. This is *strong implicit association* because it results from storage requirements defined in the ANSI standard. See example in 4.5.6, following.

Storage order is discussed in 4.5.2; array storage sequences are discussed in 4.3.6.1 and shown in figure 4-2.

4.5.5.2 Global and local data (Cray terminology)

In Cray usage, data is considered *global* if it is represented by entities in common blocks or used as arguments, because such data can be referenced and changed by more than one program unit. *Local* data is declared and accessed only by one program unit.

The ANSI standard does not define these terms as just described but makes the same distinction, such as in requirements for the SAVE statement. The distinction is important in Cray usage, for such techniques as vectorizing and multiprocessing. In the ANSI standard these terms apply only to the scope of symbolic names (see 2.1.5); for example, a variable name is local because the scope in which it can be referenced is a single program unit. In Cray usage, however, a *local variable* not only has a local name but also references local data; that is, it is not used as an argument and is not contained in a common block.

4.5.6 EQUIVALENCE STATEMENT

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

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

nlist List of two or more variable names, array element names, character substring names, or array names, separated by commas. *nlist* cannot include names of subprogram dummy arguments, pointees, or variable names that are also function names. An array name with no subscript refers only to the array's initial address.

Examples:

```
EQUIVALENCE (ARRAY1,VECTOR1), (ARRAY2,VECTOR2)
EQUIVALENCE (A(1),X), (A(73),Y), (A(247),Z)
```

An EQUIVALENCE statement specifies that the storage sequence of each entity in a list *nlist* shares the same first storage unit. This associates all entities in the list and can also indirectly associate other entities, such as adjacent addresses (higher or lower) when two array elements are associated. If entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence.

Associated entities are assigned to the same kind of storage, static or stack. Stack storage is used if **-a stack** is in the **cft77** command or ALLOC=STACK is in the CFT77 control statement (see section 1) and the entities have not been otherwise assigned to static storage (for example, with a DATA statement).

The ANSI Fortran Standard does not specify storage allocation methods.

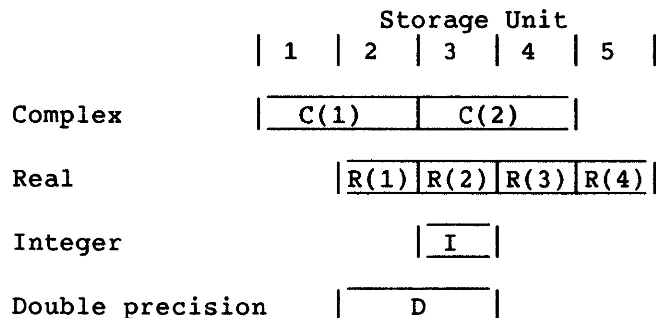
Example:

```
INTEGER I
REAL R(4)
COMPLEX C(2)
DOUBLE PRECISION D
EQUIVALENCE (C(2), R(2), I), (R,D)
```

The above EQUIVALENCE statement specifies that the following storage units are the same:

- The third storage unit of C (that is, the first unit of the second element of C)
- The second storage unit of R
- The storage unit of I
- The second storage unit of D

The storage sequences can be illustrated as follows:



As the preceding diagram shows, 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. Association of element C(1) is implied by the association of C(2) with R(2), even though C(1) does not appear in the EQUIVALENCE statement.

4.5.6.1 Array names in EQUIVALENCE statements

The use of an array name in an EQUIVALENCE statement has the same effect as using the name of the array's first array element. If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be less than or equal to the number of dimensions in the array declarator for the array. When the number of subscripts is less than the number of dimensions, the lower bounds are used for the unspecified subscripts; this does not conform to the ANSI Fortran standard, and a warning is issued.

4.5.6.2 Restrictions on EQUIVALENCE statements

An EQUIVALENCE statement must not specify the same storage unit to occur more than once in a storage sequence. Example:

```
DIMENSION A(2)
EQUIVALENCE (A(1),B) , (A(2),B)         !Illegal statement
```

The above sequence is prohibited because it specifies the same storage unit for A(1) and A(2).

An EQUIVALENCE statement must not specify consecutive storage units to be nonconsecutive. For example, the following is prohibited:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE (A(1),D(1)),(A(2),D(2))     !Illegal statement
```

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit. For example, the following is prohibited:

```
COMMON/A/X
COMMON/B/Y
EQUIVALENCE (X,Y)                       !Illegal statement
```

An EQUIVALENCE statement must not extend a common block storage sequence by adding storage units preceding the first storage unit in the block. (This unit is the first entity specified in a COMMON statement for the common block.) Example:

```
COMMON /X/A
REAL B(2)
EQUIVALENCE (A,B(2))                    !Illegal statement
```

The above sequence is not permitted because it would associate an array element B(1) with a storage unit preceding A in common block X.

An entity of type character can be equivalenced only with other entities of type character. Lengths are not required to be the same. Partial overlapping between character entities can occur through equivalance association.

Example:

```
CHARACTER A*4,B*4,C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

The above sequence partially associates A with C(2) as shown in the following illustration.

```
Character number:     |01|02|03|04|05|06|07|
                      |---- A ----|
                               |---- B ----|
                      |--C(1)--|--C(2)--|
```

4.6 COMMON BLOCKS

A *common block* is an area of memory that can be referenced by any program unit in a program, allowing a subprogram to process data as needed by another program unit. A common block can be referenced by any program unit that declares the common block.

The COMMON statement declares a common block's name (if any) along with the names of variables and arrays contained in the block. The order in which these names appear within each COMMON statement determines how the entities are associated across program units. TASK COMMON and LOCAL COMMON work the same way for common blocks used in multitasking (see 4.6.6 and 4.6.7).

Example:

```
PROGRAM EXBLOK
COMMON /TIME/ ARRAYA(10),ARRAYB(20),FACTOR
...
CALL SWING
...

SUBROUTINE SWING
COMMON /TIME/ VECTORA(10),VECTORB(20),AMULT
...
```

The two COMMON statements above declare a common block named TIME and associate VECTORA with ARRAYA, VECTORB with ARRAYB, and AMULT with FACTOR. This allows subroutine SWING to process the current values of entities appearing in the main program without the use of arguments.

Because character and noncharacter data use different kinds of storage units (see 4.5.1 and G.5), a common block must include either no characters or all characters.

A *named* common block has a name specified in a COMMON, TASK COMMON, or LOCAL COMMON statement, along with the names of variables or arrays stored in the block. A *blank* common block, often called simply "blank common," is declared in the same way but with no name shown. Blank common cannot be initialized before run time, such as with a DATA statement. You can use blank common to manage memory for your own requirements, as described in appendix D.

4.6.1 FEATURES AND UTILITIES FOR USING COMMON BLOCKS

This subsection describes compiler features for managing common blocks, and the FTREF utility.

If all program units in a program declare the same common block in the same way, entities in the common block are effectively global, as in Pascal usage. However, this global property is lost unless every change in the declaration is made in all program units. You can make global changes easily if the declaration is contained only in a separate file and inserted in each program unit by an INCLUDE statement (see 1.5).

The optional Symbol Cross-reference Table in your program listing indicates the storage type for each symbol; this includes the name of any common block containing a symbol. To get this table, include **-e sx** on the **cft77** command or ON=SX on the CFT77 control statement.

To obtain a list of all references to entities in common blocks, use the FTREF utility; this is described in the Performance Utilities Reference Manual for either UNICOS or COS, respectively publications SR-2040 and SR-0146. You can invoke FTREF as follows; the loader is not invoked because FTREF performs only a static analysis of source code.

Under UNICOS:

```
cft77 -e sx hello.f
ftref -c full -t full hello.l > hello.ref
```

Under COS (JCL statements in $IN, following JOB and ACCOUNT statements):

```
CFT77,ON=SX,L=LISTX.
FTREF,I=LISTX,CB=FULL,TREE=FULL.
/EOF
```

As specified above, the two **full** options for FTREF give the information most frequently needed: common block references and a calling tree. Under COS, you must specify a nondefault listing dataset (here LISTX) in the invocations of both CFT77 and FTREF.

4.6.2 COMMON STATEMENT

The COMMON statement specifies entities that are contained in a common block. These entities are local to the program unit containing the COMMON statement and become associated with those declared within other program units for the same common block. This allows different program units to share storage units and to define and reference the same data.

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

cb Common block name; global name of 1 to 8 alphanumeric characters, the first of which must be a letter. The blank (unnamed) common block is specified when *cb* does not appear between slashes.

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

The entities occurring in *nlist* following block name *cb* are declared to be in common block *cb*. If the first *cb* is omitted, its enclosing slashes are optional and all entities in *nlist* are specified to be in blank common.

Any *cb* (or an omitted *cb* for blank common) can occur more than once in one or more COMMON statements in a program unit. The *nlist* following each successive appearance of the same common block name continues the preceding list for that common block name.

An EQUIVALENCE statement must not extend a common block storage sequence by adding storage units preceding the first storage unit in the block. (This unit is the first entity specified in a COMMON statement for the common block.) See example in 4.5.6.2.

4.6.3 REFERENCING COMMON BLOCKS

Because a common block associates entities by storage sequence rather than by name, the names and types of variables and arrays can differ across program units. To be referenced, an entity in a common block must be defined, and the reference must be of the type which was declared for that entity in the subprogram where the reference appears. Qualifications:

- An integer variable that has been assigned an executable statement label by an ASSIGN statement must not be referenced in any program unit other than the one in which it was assigned.

- Either part of a complex entity can be referenced as a real entity.
- If any entity in a common block is of type character, all entities in the block must be of type character, and the common block definitions in all subprograms must be of type character.

A common block name may also be the name of any local entity, including a constant, intrinsic function, or a local variable that is also a function result. If a name is used for both a common block and a local entity, the name identifies only the local entity in any context other than a COMMON or SAVE statement.

The ANSI Fortran Standard does not allow a common block name to be the name of a constant, intrinsic function, or external procedure.

In a subprogram that has declared a named or blank common block, the entities in the block remain defined after the execution of a RETURN or END statement.

The ANSI Fortran Standard specifies that variables in a named common block become undefined on execution of a RETURN or END statement if no executing program unit has declared the common block.

4.6.4 COMMON BLOCK STORAGE SEQUENCE

For each common block, a *common block storage sequence* is formed as follows.

- A storage sequence is formed, consisting of the storage sequences of all entities listed in a COMMON statement. The sequence order is determined by the order of these entities.
- This storage sequence is extended to include all storage units of any storage sequence associated with it by an EQUIVALENCE statement. The sequence can be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

For all declarations of a given common block within an executable program (including blank common), the storage sequences are counted from a single beginning storage unit. That is, all entities in a declaration are positioned sequentially in relation to this starting address. The practical effects of this method of positioning are discussed in the following paragraph and in 4.6.5.

When any two local entities are to be associated across program units by means of a common block, each must be preceded by the same amount of storage within the block, as declared in the COMMON statement. Even if one of the program units does not access part of the block (in a lower address range than the entities to be associated), that program unit's COMMON statement must still declare the unused storage, such as by the use of a space-filling array.

Examples:

(1)

```
      PROGRAM FILLER
      REAL ARRAYC(10), ARRAYF(20)
      COMMON /MARKET/ A, B, ARRAYC, D, E, ARRAYF
      ...

      SUBROUTINE TARIFF
      REAL NOTHING(12), ARRAYR(20)
      COMMON /MARKET/ NOTHING, P, Q, ARRAYR
      ...
```

Subroutine TARIFF above does not reference the first 12 storage units of common block MARKET, so array NOTHING is declared to account for that storage. This allows P to be associated with D, and so on.

(2)

```
      PROGRAM INDEX
      REAL VECTOR1(100), VECTOR2(200)
      COMMON /PLACE/ VECTOR1, VECTOR2, S, T, U
      ...

      SUBROUTINE NORMAL
      REAL AINDEX(303)
      COMMON /PLACE/ AINDEX
      EQUIVALENCE (AINDEX(  1),A), (AINDEX(101),B), (AINDEX(201),C),
     &            (AINDEX(301),D), (AINDEX(303),RESULT)
      ...
```

Because subroutine NORMAL will reference only a few locations in common block PLACE, array AINDEX is used as a means of specifying locations within the block. The EQUIVALENCE statement uses the element subscripts of AINDEX to locate variables within the common block for the correct association.

4.6.5 COMMON BLOCK SIZE

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

The declarations for a given common block, appearing in different program units, are not required to declare the same amount of storage. Each declaration represents storage beginning at the block's first storage unit and equaling the total size of the entities shown in the declaration. However, a declaration of that block in any given program unit is not required to account for storage beyond the highest address referenced by the program unit.

Example:

```
      PROGRAM UNEQUAL
      REAL ARRAYV(10), ARRAYZ(20)
      COMMON /GOOD/ T, U, ARRAYV, W, X, ARRAYZ
      ...

      SUBROUTINE BENTHAM
      REAL ARRAYH(10)
      COMMON /GOOD/ F, G, ARRAYH
      ...
```

Common block GOOD declared above is allocated 34 storage units, based on its declaration in the main program. F, G, and ARRAYH are associated with T, U, and ARRAYV, respectively. Subroutine BENTHAM does not reference entities in the higher addresses in this block; its declaration does not need to account for the space, because entities are associated based only on their distance from the starting address.

The amount of storage allocated for a common block depends on the loader being used. LDR allocates the size of the common block's first declaration; this constitutes a maximum size for all subsequent declarations for that block. SEGLDR allocates the maximum size declared for a block in any program unit; therefore you do not have to stay within the size of the first declaration.

The ANSI Fortran Standard does not include variable sizes for named common blocks.

In the simplest case (and always on CRAY-2 systems), storage for blank common is statically allocated, similarly to the way it is allocated for named common blocks. If you need dynamic storage, the recommended method is the heap; see appendix D.

4.6.6 TASK COMMON STATEMENT (CFT77 EXTENSION)

When multitasking is used, some common blocks may need to be local to a task. The TASK COMMON statement declares all variables in a common block to be 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.

Keyword TASK must precede keyword COMMON to establish a task common block. Task common blocks must be named. A task common block is allocated at task invocation.

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

cb Task common block name

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

The variables in *nlist* cannot appear in a DATA statement and cannot be used in NAMELIST I/O. With these exceptions, the variables can be used like the other variables declared in common storage.

Stack allocation must be used with task common blocks; with static allocation, all task common blocks are treated as regular common blocks. (Stack storage is used if **-a stack** is in the **cft77** command or ALLOC=STACK is in the CFT77 control statement; see section 1. See 4.5.2 concerning storage modes.)

The ANSI Fortran Standard does not provide for task common blocks.

4.6.7 LOCAL COMMON STATEMENT (FOR CRAY-2 SYSTEMS)

The LOCAL COMMON statement assigns the contents of a named common block to Local Memory on the CRAY-2 computer system. Blank common cannot be declared as local common.

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

cb Local common block name

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

Local common block variables cannot be passed as actual arguments to subprograms. The SAVE and EQUIVALENCE statements treat variables in local common the same as variables in common blocks. The DATA statement cannot initialize variables in local common.

The LOCAL COMMON statement is intended for use on CRAY-2 computer systems; on other Cray computer systems, the statement is equivalent to the TASK COMMON statement.

The ANSI Fortran Standard does not provide for local common blocks.

4.6.8 BLOCK DATA SUBPROGRAM

A *block data subprogram* provides, at compile time, initial values for variables and array elements in named common blocks. In standard Fortran, other program units cannot include DATA statements to initialize common block entities, because different program units could declare conflicting data. Because of this limitation, the block data subprogram allows initialization outside of other program units. CFT77 does not include this restriction on the DATA statement, so the block data subprogram is not required.

A block data subprogram contains no executable statements and is not called by another program unit. A block data subprogram can initialize more than one common block, and one common block can be initialized in more than one block data subprogram.

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

During one invocation of CFT77, up to 26 unnamed block data subprograms can be encountered. CFT77 assigns the name BLCKDATA to the first unnamed block data subprogram, BLCKDATB to the second, BLCKDATC to the third, and so on. Separate compilations can give the same name to two different block data subprograms; this prevents proper loading of the routines, so be careful to prevent such duplication. Any number of differently named block data subprograms can be specified in an executable program.

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

4.6.6.1 BLOCK DATA statement

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

BLOCK DATA [*sub*]

sub Symbolic name of the block data subprogram in which the BLOCK DATA statement appears. *sub* is a global name and must not be the same as any other global name or any local name within the same subprogram.

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

Example:

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

EXPRESSIONS AND ASSIGNMENT 5

An *expression* specifies either: a computation involving two or more *operands* with one or more *operators*; or, one operand and an optional + or -. Operands can be constants, symbolic names of constants, variables, array elements, arrays, array sections, substrings, or function references. Operators can specify arithmetic, assignment, character, relational, or logical operations. A *constant expression* is either an arithmetic constant expression (see 5.1), a character constant expression (see 5.2), or a logical constant expression (see 5.4).

Assignment statements, for defining variables and array elements during program execution, are arithmetic, logical, and character. Table 5-1 shows which data types can be used together in an assignment statement. The ASSIGN statement, for assigning statement labels, is described in 6.4.4.

See 2.4 concerning the use of functions in expressions. Array expressions, a CFT77 extension, are discussed in 4.3.11.

An operand (a variable, array element, array, array section, or function referenced in an expression) must be defined at the time the reference is executed. Any named constant must be established in a PARAMETER statement preceding the statement where the constant is first referenced.

A parenthesized expression is treated as an entity. For example, in the expression A*(B*C), the product of B and C is evaluated and then multiplied by A. Because one operator cannot immediately follow another, parentheses are used to render a mathematical expression such as $a \cdot -b$ as A*(-B). Parenthesized expressions can be nested within other parenthesized expressions.

Precedence among all kinds of operators is as follows:

Operator	Precedence
Arithmetic	Highest
Character	⋮
Relational	
Logical	Lowest

An expression can contain more than one kind of operator. For example, the following logical expression contains arithmetic, relational, and logical operators:

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

The above expression would be interpreted the same as the following expression:

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

The compiler uses the following considerations to interpret an expression. Shown in descending order of priority, these considerations determine the order in which pairs of primaries are combined using operators.

(1) Use of parentheses

(2) Precedence of arithmetic, character, relational, and logical operators

(3) Right-to-left interpretation of exponentiations in a factor

(4) Left-to-right interpretation of all other operators

Once an interpretation has been established, CFT77 may evaluate a mathematically equivalent expression.

5.1 ARITHMETIC EXPRESSIONS

An *arithmetic expression* specifies a numeric computation. Its evaluation produces a scalar or an array.

The simplest form of an arithmetic expression is an unsigned constant or the symbolic name of a constant, variable, array element, array, array section, or function. More complicated arithmetic expressions are formed by using one or more arithmetic operands with arithmetic operators and parentheses. As shown in table 5-3, arithmetic operands are normally of type integer, real, double-precision, or complex; types Boolean and pointer can also be used, with the qualifications shown in the table.

An *arithmetic constant expression* is an arithmetic expression that contains as operands any combination of arithmetic constants, symbolic names of arithmetic constants, or arithmetic constant expressions. Exponents (y in expression x**y) must be of type integer. References to variables, array elements, or functions are not permitted. Arithmetic constant expressions are required in PARAMETER statements.

An *integer constant expression* is an arithmetic constant expression in which each constant or name of a constant is of type integer. Variables, array elements, and function references are not allowed. Integer constant expressions are required in substring designators (see 3.7.2). The ANSI

Fortran standard requires integer constant expressions in array dimension bound expressions, but CFT77 allows exceptions to this rule (see 4.3.5).

Expressions that raise 0 to a 0 or negative power or expressions that raise a negative value to a noninteger power cause run-time faults.

5.1.1 ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement defines the entity on the left of the equal sign, *v*, to be the value of the expression on the right, *e*, whose type is converted to that of *v* if required. Table 5-1 shows which combinations of operands are legal and which require conversion.

```
v = e
```

v Name of a variable, array, array section, or array element of type integer, real, double-precision, or complex

e Arithmetic expression

If *v* is a scalar, *e* must be a scalar. If *v* is an array name or array section name, the statement is an arithmetic array assignment statement and *e* must be conformable with *v* (see 4.3.11.1). If *e* is a scalar and *v* is an array name or array section name, the scalar value is assigned to all elements of the array or array section.

Examples:

```
L = 12
C = (0.8,16.5) - (16.32,-6.1)
X = -B +(B**2-4*A*C)**0.5
ROOT = SQRT(65536.0)
ARRAY(6,2,1) = 0
MATRIX(I,J,K) = MATRIX(I,J,K)+1
MATRIX(I,:,K) = MATRIX(I,:,K)+1          !Non-ANSI
ARRAY(:,:,:) = 0                         !Non-ANSI
```

The last two examples above make use of array sections and array expressions (see 4.3.10 and 4.3.11).

Table 5-1. Allowed Assignment Statements: y=x

y=x	y=I	y=R	y=D	y=C	y=B	y=P	y=L	y=S
I=x	I=I	I=$\underline{R}$[2]	I=$\underline{D}$	I=$\underline{C}$	I=B	I=P	I■L	I■S[1]
R=x	R=$\underline{I}$	R=R	R=$\underline{D}$	R=$\underline{C}$	R=B	R■P	R■L	R■S[1]
D=x	D=$\underline{I}$	D=$\underline{R}$	D=D	D=$\underline{C}$	D■B	D■P	D■L	D■S
C=x	C=$\underline{I}$	C=$\underline{R}$	C=$\underline{D}$	C=C	C■B	C■P	C■L	C■S
P=x	P=I	P■R	P■D	P■C	P=B	P=P	P■L	P■S
L=x	L■I	L■R	L■D	L■C	L■B	L■P	L=L	L■S
S=x	S■I	S■R	S■D	S■C	S■B	S■P	S■L	S=S

There are no Boolean variables, so Boolean cannot appear on the left of the equal sign.

Legend:

= The assignment is legal.

_ An underscored letter indicates the x expression on the right side of the equal sign is converted to the type of the variable on the left side.

■ The expression on the right cannot be assigned to the variable on the left.

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character	P	Pointer

1 If S is a literal character string and length (S) ≤ 8 characters, the assignment is performed with S treated as a Hollerith constant; a non-ANSI warning message is issued. See 5.2.

2 If the assignment is of the form I = X/Y, where I is integer and X and Y are real, the integer result is sometimes one less than it should be. This can occur because division on Cray systems can result in a real value ending in .99999; this fraction is lost in the integer conversion. The NINT function is one way to prevent this problem.

5.1.2 ARITHMETIC OPERATORS

Arithmetic operators and their interpretations are shown in table 5-2. Each arithmetic operator operates on a pair of operands and appears between them. The operators + and - can also operate on a single operand when it precedes that operand. An operator cannot immediately follow another operator; for example, A * -B is illegal.

The interpretation of a division operation depends on the type of the operands. An *integer quotient* is the integer portion of a quotient having an integer divisor and dividend. For example, the expression -5/2 yields an integer quotient of -2.

Table 5-2. Arithmetic Operators and Their Use in Expressions

Use of Operator	Interpretation
X**Y	Exponentiate X to the power Y
X/Y	Divide X by Y
X*Y	Multiply X by Y
X-Y	Subtract Y from X
-Y	Negate Y
X+Y	Add X to Y
+Y	(Same as Y)

5.1.2.1 Precedence of arithmetic operators

In an arithmetic expression, quantities enclosed in parentheses are evaluated first. If parentheses are within parentheses, the innermost quantity is evaluated first. Then the operations are evaluated in the following order:

First: **
Second: * and /
Third: + and -

For example, in the expression -A**2, ** has precedence over - . Therefore the result of ** is used as the operand for - . Thus, -A**2 is interpreted as -(A**2).

When an expression involves two or more operations of the same precedence, their positions within the expression determine how the expression is interpreted. (The actual order of evaluation is affected by optimization.) All operators are interpreted left to right except exponentiation (**), which is interpreted right to left.

Examples:

- A+B+C is interpreted as (A+B)+C. This does not guarantee a particular order of evaluation.
- A**B**C is interpreted as A**(B**C)

Once an interpretation has been established, CFT77 may evaluate a mathematically equivalent expression. Example:

Expression	Might Be Evaluated As
A/2./5.	A * .1
1 + A + 2	A + 3
A/B/C	A/(B*C)

Differences like those shown above can affect round-off and optimization.

5.1.3 ARITHMETIC OPERANDS

An *arithmetic operand* is an entity representing a number, which can be manipulated by an arithmetic operator. Arithmetic operands can be any of the following:

- Primaries
- Factors
- Terms
- Arithmetic expressions

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

5.1.3.1 Primaries

A *primary* is the most basic unit at one level of syntax, but can itself be an expression with its own syntax. Primaries can be any of the following:

- Unsigned arithmetic constants
- Symbolic names of arithmetic constants
- Variable references
- Array element references
- Function references
- Arithmetic expressions enclosed in parentheses
- Arithmetic array references
- Arithmetic array section references

The ANSI Fortran Standard does not provide for array expressions or array sections.

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
(K+J)	Array-valued arithmetic array expression if K and J are declared as arrays and are conformable (see 4.3.11.1).

5.1.3.2 Factors

A *factor* is a sequence of one or more primaries, separated by the exponentiation operator. A factor can be any of the following:

- *primary*
- *primary ** factor*

The second form above indicates that for interpreting a factor containing two or more exponentiation operators, the primaries must be combined from right to left. For example, the factor 2**3**2 is interpreted as 2**(3**2).

5.1.3.3 Terms

A *term* is a factor or a sequence of factors separated by multiplication or division operators. A term can be any of the following:

- *factor*
- *term / factor*
- *term * factor*

The second and third forms show that a single term can include both the * and / operators. The factors are combined from left to right in interpreting a term containing two or more multiplication or division operators.

5.1.3.4 Arithmetic expressions

An *arithmetic expression* is a term or sequence of terms separated by addition (+) or subtraction (-) operators. The forms of an *arithmetic expression* are as follows:

- *term*
- + *term*
- - *term*
- *arithmetic expression* + *term*
- *arithmetic expression* - *term*

The first term in an arithmetic expression can be preceded by an identity (+) or negation (-) operator. The last two forms show that terms are combined from left to right in interpreting an arithmetic expression containing two or more addition or subtraction operators.

These formation rules prohibit expressions containing two consecutive arithmetic operators such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

5.1.4 DATA TYPE OF ARITHMETIC EXPRESSIONS

The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands. The data types of arithmetic expressions are given in table 5-3, and those for exponentiation in table 5-4. Each table item represents an expression and a result, and each capital letter represents an operand or result, as follows:

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character	P	Pointer

In table 5-3, an entry is similar in form to an assignment statement, but also applies to expressions within parentheses. The ∘ symbol represents an arithmetic operator. If an arrow is shown, it points to the result type; a square symbol with no arrow indicates the two operands cannot be used in an arithmetic expression. In each row and column, one operand has the same data type throughout, and the other operand changes; the unchanging operand is indicated in the column head and at the beginning of each row.

Example:

```
R←I∘R
```

To the right of the arrow, ∘ is an operator and I and R are operands of types integer and real. To the left of the arrow, R indicates a real result; the underscored I indicates that integers in such an expression are converted to type real. The preceding entry would apply to the following expression:

```
CTEMP * 9/5 + 32
```

This would be treated as

```
CTEMP * 9./5. + 32.
```

Note that CTEMP+(9/5)+32 is treated as CTEMP+1.+32.

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

In general, conversion of data types is determined by a hierarchy of types, and is always upward within the hierarchy. The hierarchy is as follows:

```
       Complex
         - -
  Double-precision
         - -
        Real
         - -
       Integer
```

Because type Boolean is not converted, it does not fit in the hierarchy; it is used only with types integer, real, and pointer. Pointer type is used only with types integer, Boolean, and pointer. Types character and logical are not allowed in arithmetic expressions.

In an expression operating on either a single operand or a pair of operands, the type and interpretation are independent of the context where the expression appears, and independent of the type of any other operand of a larger expression where the expression appears.

Example:

```
COMPLEX CPX
CPX = (-2.)**0.5          !Illegal
```

Although, in mathematics, complex numbers derive from operations on real numbers, the above exponentiation is illegal, as shown in table 5-4.

Table 5-3. Use of Data Types with Arithmetic Operations: +, -, *, /

	y←I∘x	y←R∘x	y←D∘x	y←C∘x	y←B∘x	y←P∘x	y←S∘x
y←x∘I	I←I∘I	R←R∘$\underline{I}$	D←D∘$\underline{I}$	C←C∘$\underline{I}$	[1]I←B∘I	[4]I←P∘I	[3]■ S∘I
y←x∘R	R←$\underline{I}$∘R	R←R∘R	D←D∘$\underline{R}$	C←C∘$\underline{R}$	[2]R←B∘R	■ P∘R	[3]■ S∘R
y←x∘D	D←$\underline{I}$∘D	D←$\underline{R}$∘D	D←D∘D	C←C∘$\underline{D}$	■ B∘D	■ P∘D	■ S∘D
y←x∘C	C←$\underline{I}$∘C	C←$\underline{R}$∘C	C←$\underline{D}$∘C	C←C∘C	■ B∘C	■ P∘C	■ S∘C
y←x∘B	[1]I←I∘B	[2]R←R∘B	■ D∘B	■ C∘B	[1]B←B∘B	[4]I←P∘B	[3]■ S∘B
y←x∘I	[4]I←I∘P	■ R∘P	■ D∘P	■ C∘P	[4]I←B∘P	[4]I←P∘P	■ S∘P
y←x∘S	[3]■ I∘S	[3]■ R∘S	■ D∘S	■ C∘S	[3]■ B∘S	■ P∘S	[3]■ S∘S

Legend:

x	One of two operands. A capital letter represents another operand of the indicated data type. Logical is never allowed.
∘	Arithmetic operator: + - * / . Two operands and an operator are an expression.
y	Result of arithmetic operation. A letter to the left of the ← symbol represents a result of the indicated data type.
■	Error
_	An underscored operand's data type is converted before computation of the expression.

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	P	Pointer	S	Character		

1 Integer operation; no conversion is performed on the Boolean item.

2 Real operation; no conversion is performed on the Boolean item.

3 If S is a literal character string and length (S) $\leq$ 8 characters, then the operation is performed with S treated as Hollerith; a non-ANSI warning message is issued. See 5.2.

4 The only allowed expressions are: P+I, P-I, I+P, I-P, P+B, P-B, B+P, B-P, P+P, and P-P. All allowed expressions produce integer results. Multiplication and division of pointers are not allowed.

The ANSI Fortran Standard does not allow complex and double-precision types to be mixed in arithmetic operations.

Table 5-4. Data Types in Exponentiation: **

	y←I**x	y←R**x	y←D**x	y←C**x
y←x**I	I←I**I	R←R**I	D←D**I	C←C**I
y←x**R	R←I**R	R←R**R	D←D**R	C←C**R
y←x**D	D←I**D	D←R**D	D←D**D	[1]C←C**D
y←x**C	C←I**C	C←R**C	C←D**C	C←C**C

Types Boolean, logical, character, and pointer cannot be used in exponentiation.

In an expression a**b, if a is negative, b must be of integer type.

Legend:

x One of two operands. A capital letter represents an operand of the indicated data type.

** Exponentiation operator

y Result of exponentiation operation. A letter to the left of the ← symbol represents a result of the indicated type.

■ Error

_ Underscore indicates that the indicated operand's data type is converted before the operation is performed.

I Integer R Real D Double-precision C Complex

1 The double-precision exponent is converted to type real.

The ANSI Fortran Standard does not allow complex and double-precision types to be mixed in exponentiation operations.

5.1.4.1 Type conversion

Type conversion of operands can occur during an expression's evaluation or when the results of an expression's evaluation are stored into a variable or array element. Type conversion is based on the following two operations.

(a) Integer-to-real conversion creates a real value from an integer value.

(b) Real-to-integer conversion creates a 64-bit integer value from a real value. The fractional part is truncated.

In the above conversions, the absolute value of the integer must be less than 2**46, or less than 2**63 if 64-bit integer arithmetic is enabled. A warning is issued if the value is determined at compile time to exceed this range, but not for an out-of-range value that occurs at run time.

Integer - Type integer is converted to type real as described in item (a) above. Integer is converted to double-precision by converting to real and adding zeros to extend the precision of the value. For converting integer to complex, the integer is converted to real as described in item (a) above; then the integer value becomes the real portion of a complex value, and zero is the imaginary portion.

Real - Real is converted to integer as described in item (b) above. For converting to double-precision, zeros are added as the least significant portion to extend the real number's precision. For converting real to complex, the real value becomes the real portion of a complex value and zero is the imaginary portion.

Complex - Complex is converted to integer by converting the real portion of the complex value as described for the real value in item (b) above. For converting complex to real, the real portion of the complex value becomes the real value. For converting complex to double-precision, zeros are added to extend the precision of the real portion of the complex value; this extended real portion becomes the double-precision value.

Double-precision - Double-precision is converted to integer by converting the most significant portion as in item (b) above. For converting to real, the most significant portion of the double-precision value is used as the real value; no rounding occurs. For converting to complex, the most significant portion of the double-precision value becomes the real portion of the complex value, and zero is the imaginary portion; no rounding occurs.

The ANSI Fortran Standard does not provide for converting double-precision to complex.

5.1.5 CONSIDERATIONS IN EVALUATING ARITHMETIC EXPRESSIONS

In expressions that include integers, truncation can cause unintended results. Example:

```
(3.0*10)/3  equals  30.0/3  equals  10.0

3.0*(10/3)  equals  3.0*3  equals  9.0
```

Because the expression 10/3 gives an integer quotient, the placement of parentheses in the two expressions above changes the result.

In addition to truncation in the use of integers, results can be affected by round-off errors and finite approximation of real numbers. For example, the difference between 5./10. and 5.*.1 is a computational difference; that is, although the two expressions are mathematically equivalent, the resulting values in a computer program could be different. Predicting and controlling computational differences is beyond the scope of this book but is discussed in various textbooks.

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

Example:

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

The term (B-C) above is evaluated and then added to A. Removing parentheses could change the computed value.

5.2 CHARACTER EXPRESSIONS

A *character expression* is one character primary or a sequence of character primaries joined by the concatenation operator // . A *character constant expression* is a character expression in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses.

A *character primary* is one of the following:

- A character constant or the symbolic name of a character constant
- A character substring
- A variable, array element, or function reference of type character
- A character expression enclosed in parentheses

- Character array reference
- Character array section reference
- Character array substring reference
- Character array section substring reference

See 5.3 concerning relational expressions involving character operands.

To convert a numeric value to a character string that would print out as that value, write the value to a character variable used as an internal file (see 7.4). Example:

```
INTEGER I
CHARACTER CHVAR
WRITE (CHVAR,'(I5)') I
```

Following the above WRITE statement, character variable CHVAR contains the digits from integer variable I, allowing the number to be processed as a character string.

5.2.1 CHARACTER ASSIGNMENT STATEMENT

Execution of a character assignment statement causes the evaluation of a character expression *ce* and the definition of character entity *cv* with the value of *ce*.

cv = *ce*

cv Name of a character variable, array element, whole array, array section, substringed array section, substring, or substring of a whole array. See 3.7.2 concerning substrings of array elements.

ce Character expression

Where appropriate, *ce* is either truncated or padded with blanks on the right to match the length of *cv*. No character positions defined in *cv* can be referenced in *ce*.

5.2.2 CHARACTER EXPRESSION EVALUATION

The result of evaluating a character expression is always of type character. Primaries are combined from left to right. Example:

```
      CHARACTER*2 VAR1,VAR2
      VAR1='CR'
      VAR2='AY'
        ...
      PRINT *,VAR1//VAR2
```

The preceding sequence produces the following printed result:

```
CRAY
```

5.2.3 HOLLERITH TYPE

The Hollerith data type is described in E.1. Because CFT77 supports the representation of Hollerith using the same conventions as the character type, the compiler must determine which type applies in ambiguous situations.

A character constant is treated as Hollerith in contexts where a character constant is illegal and a Hollerith constant is legal. Tables 5-1, 5-2, 5-5, and 5-8 indicate which expressions fall into this category. This "passive" Hollerith typing (that is, letting the compiler determine that a constant is Hollerith) causes a non-ANSI warning message to be issued. If you intend a constant to be Hollerith, it is suggested that you make it explicit by including an H after the second delimiter; for example 'ABC'H.

5.3 RELATIONAL EXPRESSIONS

A *relational expression* compares the values of two arithmetic or character expressions, producing a logical value of true or false. A relational expression can be assigned to a variable in a logical assignment statement (see 5.4.1). Relational expressions can appear within logical expressions. Relational operators are as follows:

Operator	Operation (comparison)
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to
.GT.	Greater than

Relational operators have no precedence within this group because the use of more than one operator in the same relational expression is illegal.

Table 5-5 shows which data types can be used together in relational operations. The result type is logical for all relational operations.

5.3.1 ARITHMETIC RELATIONAL EXPRESSIONS

An *arithmetic relational expression* is a relational expression whose operands are arithmetic expressions or arithmetic array expressions. An arithmetic relational expression is interpreted as the logical value TRUE if the values of the expressions satisfy the relation specified by the operator; FALSE if they do not. If at least one of the operands is an array, an array of logical values is returned.

Examples:

```
A .LE. B
INDEX .EQ. ENDVALU
J(1,6,6)*COS(ALPHA/10.) .GT. Z
```

In the relational expression *x relop y*, the result of the expression *x-y* must be within the range defined for the data type (see 3.2, 3.3, 3.4, and 3.5). If arithmetic expressions *i* and *r* are of different types, the expression *i relop r* is evaluated as if it were ((*i*)-(*r*)) *relop* 0, with the expression in parentheses evaluated according to the type conventions for arithmetic expressions, as shown in table 5-3. The additional parentheses affect round-off and optimization.

Example:

```
INTEGER INTG
COMPLEX CMPX
DOUBLE PRECISION DBPR
LOGICAL RELC, RELD
INTG = 2
CMPX = (2.,0.)
DBPR = 2.
RELC = INTG .EQ. CMPX        ! Tests true
RELD = INTG .EQ. DBPR        ! Tests true
```

The above code causes both logical variables RELC and RELD to be true. In the assignment of RELC, the relational expression INTG .EQ. CMPX is treated as

```
(INTG - CMPX) .EQ. 0
```

As shown in table 5-3, the arithmetic expression (INTG - CMPX) is evaluated after INTG is converted to its complex equivalent; in the example, this equivalent equals the value assigned to CMPX.

However, RELD is false in the following code:

```
INTG = 2
DBPR = 4.**0.5
RELD = DBPR .EQ. INTG          ! Tests false
```

Although 4.**0.5 nominally equals 2., it does not compute to exactly this value, and therefore does not equal the double-precision equivalent of variable INTG. On the other hand, if DBPR is set to SQRT(4.), RELD might be true; but program decisions should not be based on assumptions about exact values of computations. For real computations, a better approach is to define a range of accuracy. In the following code, RELD tests true, indicating that DBPR and INTG are within 10^{-10} of each other:

```
EPSILON=1.E-10
INTG = 2
DBPR = 4.**0.5
RELD = ABS(DBPR-INTG) .LT. EPSILON      ! Tests true
```

5.3.2 CHARACTER RELATIONAL EXPRESSIONS

A *character relational expression* is a relational expression in which both operands are character expressions. The result is interpreted as the logical value TRUE if the values of the operands satisfy the relation specified by the operator; otherwise, the result is interpreted as the logical value FALSE.

The character expression that comes first in the collating sequence (see appendix A) is considered to be of lower value. If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand.

5.4 LOGICAL EXPRESSIONS

A *logical expression* specifies a logical operation on logical operands. Evaluation of a logical expression produces a result of type logical with a value of either true or false (see 3.6). A *logical constant expression* is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses. Note that relational expressions also give logical results and, within parentheses, can appear in logical expressions. Logical values and expressions are contrasted to Boolean values and masking expressions in 3.8.

Tables 5-6 and 5-7 show the *logical operators* and their usage.

Table 5-5. Data Types in Relational Operations: .EQ.,.NE.,.GT.,.GE.,.LT.,LE.

	I	R	D	C	B	P	S
I	L	L	L	L[a]	L[c]	L[c]	■[b]
R	L	L	L	L[a]	L[c]	■	■[b]
D	L	L	L	L[a]	■	■	■
C	L[a]	L[a]	L[a]	L[a]	■	■	■
B	L[c]	L[c]	■	■	L	■	■[b]
P	L[c]	■	■	■	■	L	■
S	■[b]	■[b]	■	■	■[b]	■	L

Relational operands cannot be of type logical.

Legend:

L The relational operation is allowed. The result type is always logical.

■ Prohibited

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character	P	Pointer

a Only .EQ. and .NE. are allowed for complex comparisons.

b If S is a literal and length (S) ≤ 8 characters, the operation is performed with S treated as a Hollerith constant; a non-ANSI warning message is issued. See 5.2.

c The comparison is performed without conversion of either operand.

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

5.4.1 LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement assigns the value of logical expression *le* to logical entity *lv*.

```
lv = le
```

lv Name of a logical variable, logical array, logical array section, or logical array element

le Logical expression

If *lv* is a scalar, *le* must be a scalar. If *lv* is an array name or array section name, the statement is a logical array assignment statement and *le* must be conformable with *lv*. If *le* is a scalar and *lv* is an array name or array section name, the scalar value is assigned to all elements of the array or array section.

The ANSI Fortran Standard does not provide for array expressions or array sections.

Examples:

All variable and array element names are assumed to be of type logical except E and F, which are type real, and I, J, K, and L, which are type integer.

```
T = .FALSE.
A = B
C = (A .AND. B) .OR. (C .AND. D)
T = .NOT. T
TRUTAB(I,J,K,L) = .T.
T = E.GE.F .OR. E/F .LT. .4
T = A .EQV. B
```

5.4.2 LOGICAL OPERATORS

The logical operators are shown in table 5-6; their use with operands is shown in table 5-7. The .NOT. or .N. operator produces the logical complement of its operand.

For logical expressions containing two or more logical operators, the precedence, as shown in table 5-6, determines the order in which they are to be combined (unless changed by the use of parentheses).

Example:

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

In this expression, .AND. has higher precedence than .OR. Therefore the expression is interpreted as follows:

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

Logical operators can also be written as functions; for example A.AND.B can be written as AND(A,B). .NOT. is written as COMPL(). See table B-8.

The ANSI Fortran Standard does not provide for the function forms of logical operators.

Table 5-6. Logical Operators

Operator	Operation	Precedence
.NOT. or .N.	Logical negation	Highest
.AND. or .A.	Logical conjunction	.
.OR. or .O.	Logical inclusive disjunction	.
.XOR. or .X. or .NEQV.	Logical exclusive disjunction or logical nonequivalence	.
.EQV.	Logical equivalence	Lowest

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

Table 5-7. Meanings of Logical Operators

X1	X2	.NOT. X1	X1.AND.X2	X1.OR.X2	X1.XOR.X2 X1.NEQV.X2	X1.EQV.X2
true	true	false	true	true	false	true
true	false		false	true	true	false
false	true	true	false	true	true	false
false	false		false	false	false	true

5.4.3 FORM AND INTERPRETATION OF LOGICAL EXPRESSIONS

A *logical operand* is an entity that can be operated on by a logical operator. Logical operands can be any of the following:

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

A *logical primary* is a primary in a logical expression. Logical primaries can be any of the following:

- Logical constants
- Symbolic names of logical constants
- Logical variable or array element references
- Logical function references
- Relational expressions
- Logical expressions enclosed in parentheses
- Logical array references
- Logical array section references

A *logical factor* consists of a logical primary with or without the .NOT. operator. The form of a logical factor is:

[.NOT.] *logical-primary*

A *logical term* is a sequence of logical factors separated by an .AND. operator. If a logical term contains two or more .AND. operators, the logical factors are combined from left to right. The form of a logical term is:

[*logical-term* .AND.] *logical-factor*

A *logical disjunct* is a sequence of logical terms separated by an .OR. operator. If a logical disjunct contains two or more .OR. operators, the logical terms are combined from left to right. The form of a logical disjunct is:

[*logical-disjunct* .OR.] *logical-term*

A *logical expression* is a sequence of logical disjuncts separated by .XOR., .EQV., or .NEQV. operators. If a logical expression contains two or more .XOR., .EQV., and/or .NEQV. operators, the logical disjuncts are combined from left to right. The forms of a logical expression are:

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

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

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

5.5 MASKING EXPRESSIONS (CFT77 EXTENSION)

A *masking expression* is an expression in which a logical operator operates on individual bits within integer, real, pointer, or Boolean operands, giving a result of type Boolean. Each operand is treated as a single storage unit (a 64-bit Cray word), and the result is a single storage unit. Boolean values and masking expressions are contrasted to logical values and expressions in 3.8.

The ANSI Fortran Standard does not provide for masking expressions.

Masking operators can also be written as functions; for example A.AND.B can be written as AND(A,B). .NOT. is written as COMPL(). See table B-8. The same table shows other functions that operate on Boolean values, such as shifting, parity count, and tallying 1's or leading 0's.

Table 5-8 shows which data types can be used together in masking operations. Letters in the table indicate the result type for each allowed operation. A masking expression cannot have operands of type logical, double precision, or complex.

Masking expressions can be combined with expressions of Boolean or other types by using arithmetic, relational, and logical (masking) operators. Evaluation of an arithmetic or relational operator processes a masking expression with no type conversion. Boolean data is never converted to another type.

A logical (masking) operator processing a masking expression performs the indicated logical operation separately on each bit. The interpretation of individual bits in masking factors, terms, and expressions is the same as for logical expressions (see 5.4). The results of binary 1 and 0 correspond to the logical results TRUE and FALSE, respectively, in each of 64 bit positions. These values are summarized as follows:

```
.NOT. 1100              1100            1100             1100            1100
     =0011       .AND.  1010      .OR.  1010      .XOR.  1010     .EQV.  1010
                        ----            ----             ----            ----
                        1000            1110             0110            1001
```

Table 5-8. Allowed Logical and Masking Operations and Result Types

x \ y	I	R	B	P	L	S
I	B	B	B	B	■	■[1]
R	B	B	B	B	■	■[1]
B	B	B	B	B	■	■[1]
P	B	B	B	B	■	■[1]
L	■	■	■	■	L	■
S	■[1]	■[1]	■[1]	■[1]	■	■

Types complex and double-precision cannot be used in logical or masking operations.

Legend:

x,y Operands for a masking or logical expression, using operands .NOT., .AND., .OR., .XOR., and .EQV.

Entries in table:

B Masking operation with result type Boolean

L Logical operation with result type logical

■ Prohibited

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character	P	Pointer

1 If S is a literal and length (S) ≤ 8 characters, the operation is performed with S treated as a Hollerith constant. A non-ANSI warning message is also issued. See 5.2.

PROGRAM CONTROL 6

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, described in this section, control an execution sequence.

- Conditional block statements: IF THEN, ELSEIF, ELSE, ENDIF
- Arithmetic IF
- Logical IF
- DO
- Unconditional GOTO
- Computed GOTO
- Assigned GOTO
- STOP
- PAUSE
- END

The ASSIGN and CONTINUE statements, used in conjunction with the above, are also described in this section.

In addition to the above statements, the following statements can alter the execution sequence from the order in which they appear in a program.

- CALL (see 2.5.2.2)

- RETURN (see 2.5.3.2)

- An I/O statement containing an error specifier or an end-of-file specifier (described in section 7)

6.1 CONDITIONAL BLOCKS

A *conditional block* is a group of executable statements delimited by the following *conditional block statements*, which control execution of the blocks:

- Block IF
- ENDIF
- ELSEIF
- ELSE

A conditional block is executed if a certain condition is true; the condition is specified in an IF or ELSEIF statement as a logical expression. The statements that define conditional blocks interact as described below.

The *IF level* of a statement is the number of block IF statements from the beginning of the program unit to that statement minus the number of ENDIF statements from the beginning of the program unit up to but not including that statement. That is, the IF level is incremented by a block IF statement and decremented by an ENDIF statement.

Each block IF statement must correspond to a later ENDIF statement at the same IF level. Between a block IF and its ENDIF is a group of one or more conditional blocks; in this manual the entire range from an IF through its ENDIF is called an *IF structure*. Only one conditional block in an IF structure can be executed (at the IF level of the structure's IF statement); this is the first block whose condition is true, or, if none is true, an ELSE block if present.

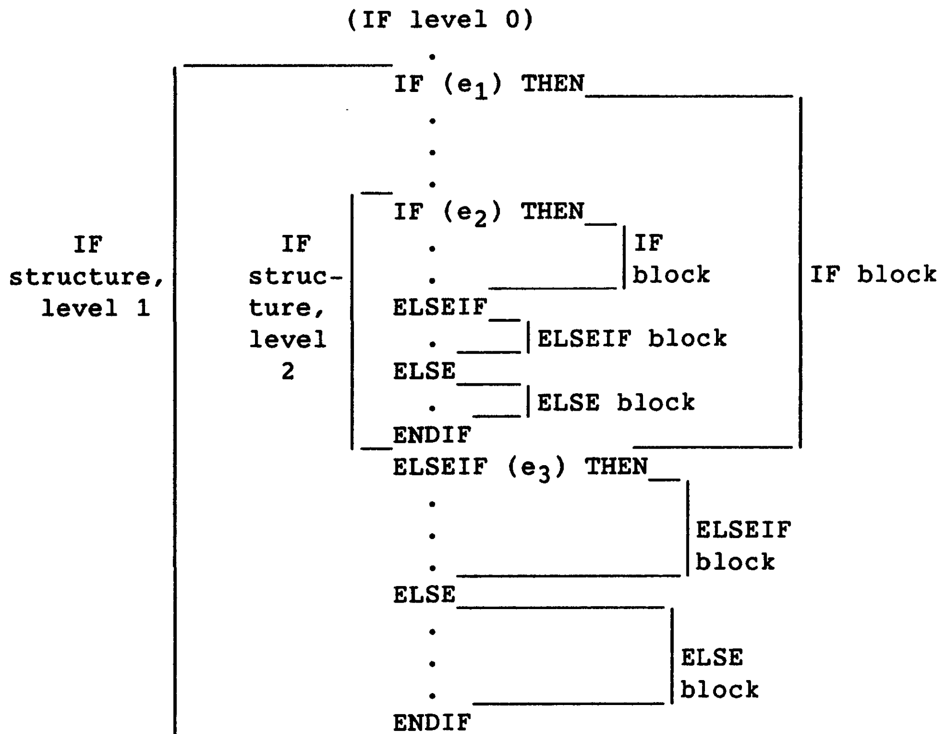
Conditional blocks are IF blocks, ELSEIF blocks, or ELSE blocks; a block begins with a block IF, ELSEIF, or ELSE statement and continues up to an ENDIF at the same IF level or the beginning of the next block. ELSEIF and ELSE blocks are optional in an IF structure. Control must not be transferred to a statement within a block from outside that block. Any conditional block may be empty.

Figure 6-1 shows the three kinds of blocks both in schematic form and in a sample of Fortran code. Notice that an IF structure is not equivalent to an IF block; an IF structure includes the IF and ENDIF statements and can include other blocks at the same IF level.

An *IF block* is a group of executable statements that are executed if the condition specified in the block IF statement is true. The block is preceded by a block IF statement and ends with (but does not include) the next conditional block statement (ENDIF, ELSEIF, or ELSE) at the same IF level.

An *ELSEIF block* is a group of executable statements that are executed if the ELSEIF's condition is true, and if no preceding block was executed in the same IF structure (either the IF block or a previous ELSEIF block at the same IF level). The block begins with an ELSEIF statement and ends with (but does not include) the next conditional block statement (ENDIF, ELSEIF, or ELSE) at the same IF level.

An *ELSE block* is a group of executable statements that are executed if no preceding block in the same IF structure (and at the same IF level) was executed. An ELSE block begins with an ELSE statement and ends with (but does not include) an ENDIF statement of the same IF level. No other conditional block statement at the same level can appear after the ELSE statement or before the ENDIF statement.



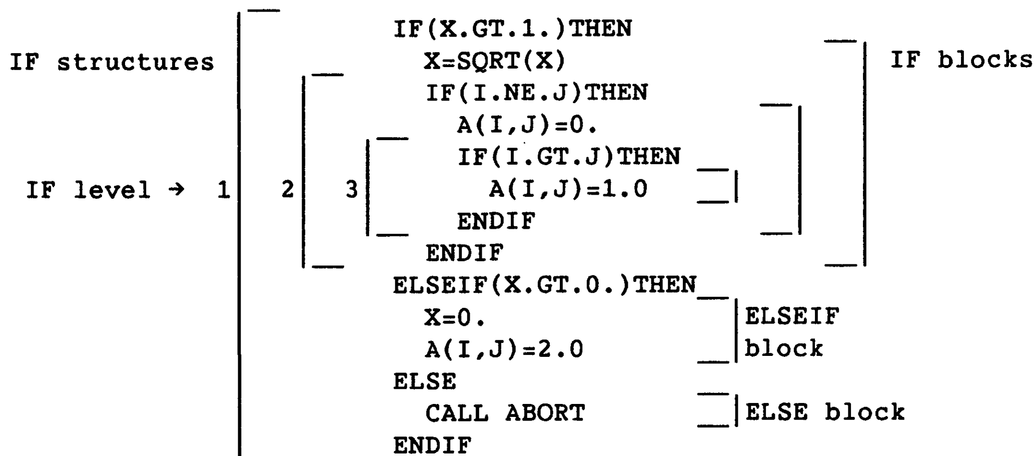


Figure 6-1. IF levels and Conditional Blocks in an IF Structure

6.1.1 BLOCK IF STATEMENT

The block IF statement determines, from the value of logical expression *le*, whether to execute the group of statements up to the next conditional block statement at the same IF level: ENDIF, ELSEIF, or ELSE. The block IF statement must always have a corresponding ENDIF statement of the same IF level.

```
IF (le) THEN
```

le Logical expression (see 5.4)

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

Transfer of control into an IF block from outside the IF block is prohibited.

6.1.2 ENDIF STATEMENT

The ENDIF statement indicates the end of an IF level and must always correspond to an earlier block IF statement of the same IF level. The ENDIF statement consists of the word ENDIF alone.

6.1.3 ELSEIF STATEMENT

The ELSEIF statement is executed if no preceding block has been executed (at the same IF level and in the same IF structure). The ELSEIF statement determines, from the value of logical expression *le*, whether to execute the group of statements up to the next conditional block statement at the same IF level: ENDIF, ELSEIF, or ELSE.

```
ELSEIF (le) THEN
```

le Logical expression (see 5.4)

Transfer of control into an ELSEIF block from outside the ELSEIF block is prohibited. Statement labels on ELSEIF statements cannot be referenced.

6.1.4 ELSE STATEMENT

The ELSE statement is executed if no previous block has been executed (at the same IF level and in the same IF structure). If it is executed, its ELSE block is executed (the group of statements up to the next ENDIF at the same IF level). The statement consists of the word ELSE alone; any statement label on an ELSE statement cannot be referenced.

6.2 OTHER IF STATEMENTS

Fortran includes two other conditional statements besides those for conditional blocks. The logical IF can replace an IF block of only one statement; the arithmetic IF transfers control based on the value of an expression. See E.5 concerning outmoded kinds of IF statements.

6.2.1 LOGICAL IF STATEMENT

The logical IF statement determines, from the value of logical expression *le*, whether to execute the statement following *le*. If the value of *le* is false, the statement is not executed and the execution sequence proceeds as though a CONTINUE statement were executed.

IF (*le*) *st*

le Logical expression (see 5.4). A function reference in *le* might affect related entities in the statement *st*.

st Any executable statement other than DO, END, block IF, ELSEIF, ELSE, ENDIF, or another logical IF statement.

Examples:

```
IF (X.GT.0) X=A
IF(K) K=.NOT.K
IF (A.EQ.B) GOTO 100
```

6.2.2 ARITHMETIC IF STATEMENT

The arithmetic IF statement transfers control to one of three statements, depending on the value of a specified expression.

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

e Integer, real, or double-precision expression. If the expression's value is less than 0, control transfers to the statement identified by s_1; if 0, statement s_2; greater than 0, statement s_3.

s_1, s_2, and s_3
Statement labels of executable statements that appear in the same program unit as the arithmetic IF statement. The same statement label can appear more than once in this statement.

Examples:

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

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

6.3 DO LOOPS

A *DO loop* consists of a DO statement and a set of statements to be executed repeatedly. The range of a DO loop consists of all executable statements from the first executable statement following the DO statement and ending with the *terminal statement* of the DO loop.

The number of times a DO loop is to be repeated is the *trip count*, which is initially determined from expressions in the DO statement, and decremented for each iteration of the loop. The *DO variable* is set to an initial value and incremented or decremented by the *increment value* until its value reaches or exceeds the *limit value*. **-e j** on the **cft77** command or ON=J in the CFT77 control statement causes at least one execution of any DO loop whose DO statement is reached.

Example:

```
      DO 20 I=1,10                 ! I is the loop's DO variable
        ARR(I) = SQRT(100.*I)      ! This statement is in the loop's range
20    CONTINUE                     ! Terminal statement
```

In the DO statement above, I is the DO variable; the DO variable's initial value is 1; its limit value is 10; and its increment value is the default of 1. The resulting trip count is 10. Notice that the DO variable I is used within the loop.

A DO loop can appear within another DO loop and must be entirely contained within the outer DO loop range. More than one DO loop can have the same terminal statement, but separate terminal statements improve programming clarity.

A DO loop can appear within a conditional block but must be entirely contained within that block. If a block-IF statement appears within the range of a DO loop, the corresponding ENDIF statement must appear within the same DO loop.

A DO loop is either active or inactive. A DO loop is initially inactive and becomes active only when its DO statement is executed. Control must not transfer into the range of an inactive DO loop. An active DO loop becomes inactive under any of the following conditions.

- Its trip count is tested and determined to be zero.
- A RETURN statement is executed within the DO loop.
- A STOP statement is executed, or program execution is terminated in any other way.
- Control is transferred outside the DO loop within the same program unit.

When a DO loop becomes inactive, the DO variable is unaffected: it retains its last defined value or remains undefined.

6.3.1 DO STATEMENT

A DO statement specifies necessary information to control the repeated execution of a set of statements.

```
DO label [,] dvar = init,lim[,incr]
```

label — Statement label of the loop's *terminal statement,* which must be executable

dvar — Name of the DO variable; must be a simple variable of type integer, real, double-precision, or pointer. *dvar* is initially defined with *init* and is incremented by *incr* on each iteration of the loop. Within the range of a loop, *dvar* can be used but cannot be redefined, including by a nested DO statement.

init, lim, and *incr* — Initial value, limit, and increment of the DO variable; must be integer, real, or double-precision expressions. If necessary, types are converted to the type of the DO variable, according to the rules for arithmetic conversion. *incr* defaults to 1 and cannot equal 0.

init, lim, and *incr* can be redefined with no effect on loop control processing. **-e j** in the **cft77** command or ON=J in the CFT77 control statement causes at least one execution of each DO loop.

The initial trip count is established as the integer portion of the following expression, or as zero if either condition shown in the next paragraph is true. (*incr* cannot equal 0.)

(*lim-init+incr*) / *incr*

The initial trip count must be less than 2^{32}-1 on the CRAY-2 computer system and less than 2^{24}-1 on other Cray systems.

Execution of the loop ends when the trip count is zero, which occurs when either of the following conditions is true:

- *init* > *lim* and *incr* > 0
- *init* < *lim* and *incr* < 0

If either of the above conditions is true initially, the loop is not executed and a message is issued; see example 2 in 6.3.3.

6.3.2 TERMINAL STATEMENT AND CONTINUE STATEMENT

The *terminal statement* is an executable statement that ends the DO loop. The statement can be executed in the normal sequence, or control can be transferred to it. The statement must not be a block IF, ELSEIF, ENDIF, arithmetic IF, unconditional GOTO, assigned GOTO, RETURN, STOP, END, or another DO statement. It can be a logical IF statement.

The CONTINUE statement is commonly used as a terminal statement; it has no effect. As with any terminal statement, the next statement executed depends only on the result of DO loop processing.

Examples:

1)

```
      DIMENSION ARRAY(16)
      ...
      DO 10,I=1,16
        IF(ARRAY(I).NE.0) ARRAY(I)=1.0/ARRAY(I)
   10 CONTINUE
```

The above loop replaces nonzero elements of ARRAY with their reciprocals. The initial trip count is (16-1+1)/1 or 16, the number of elements in the array.

2)

```
      DIMENSION TABLE(50)
      ...
      DO 10 I=1,49
        IF(TABLE(I).LE.0) THEN
          TABLE(I)=-TABLE(I)
        ELSE
          TABLE(I)=-TABLE(I+1)
        ENDIF
   10 CONTINUE
```

The above loop uses an IF block to determine the effect of each iteration.

3)

```
      INTEGER A(10,9)
      DO 10 I=1,10
        DO 9 J=1,9          !Nested loop
          A(I,J)=I*J
    9 CONTINUE
   10 CONTINUE
```

The above example shows a loop nested within another loop, which is used for a two-dimensional array.

6.3.3 LOOP CONTROL AND INCREMENTATION PROCESSING

Loop control processing determines if further execution of a DO loop is required. If the trip count is not 0, control transfers to the first statement in the range of the DO loop. If the trip count is 0, the loop becomes inactive, and control is transferred to the first executable statement after the terminal statement (provided that other DO loops sharing the same terminal statement are also inactive; otherwise execution resumes with incrementation processing, but without execution of the terminal statement). After the terminal statement executes, incrementation processing occurs unless the terminal statement results in a transfer of control.

Incrementation processing consists of the following sequence.

1. The value of the DO variable is incremented by the value of *incr*.

2. The trip count is decremented by 1.

3. Execution continues with loop control processing of the same DO loop whose trip count was decremented.

A DO variable can increase or decrease in value during incrementation processing.

Examples:

1) For discussion only; NOT RECOMMENDED

```
      M=0
      DO 10 I=1,10
        J=I
        DO 10 K=1,5
          L=K
   10 M=M+1                   !Combined terminal stmt to show processing
```

After the last execution of the final statement above, I=11, J=10, K=6, L=5, and M=50.

2) For discussion only; NOT RECOMMENDED

```
      N=0
      DO 20 I=1,10
        J=I
        DO 20 K=5,1           !Specification prevents loop execution
          L=K
   20 N=N+1                   !Combined terminal stmt to show processing
      CONTINUE
```

The nested loop above does not execute because of the specification K=5,1. (A specification like this causes a message to be issued.) Because the nested loop's trip count is always 0, statement 20 is never executed, but merely activates incrementation processing for the outer loop. Therefore, at execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by these statements because it is in the nested loop.

6.4 GOTO AND ASSIGN STATEMENTS

GOTO statements specify other statements within the same program unit to which control is transferred.

6.4.1 UNCONDITIONAL GOTO STATEMENT

The unconditional GOTO statement transfers control to the statement identified by the statement label.

```
GOTO s
```

s Statement label of an executable statement in the same program unit.

Examples:

```
      GO TO 845
      GOTO 910
```

6.4.2 COMPUTED GOTO STATEMENT

The computed GOTO statement transfers control to the statement identified by the *i*th statement label in a list, when *i* is the value of integer expression e. If *i* is less than 1 or more than the number of statement labels, the execution sequence proceeds as if the GOTO were a CONTINUE statement.

```
GOTO (s[,s]...)[,]e
```

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

e Integer expression

Examples:

```
GOTO (0031,59,728)IX
GOTO (0031,59,728)MSIZE/2
GOTO (6,3,6,6,7,2,7),NBRANCH
```

6.4.3 ASSIGNED GOTO STATEMENT

The assigned GOTO statement transfers control to the statement identified by variable *i*. *i* is defined with a statement label by an ASSIGN statement in the same program unit (see 6.4.4). The label must be for an executable statement and *i* must be defined when the assigned GOTO is executed. The statement following the assigned GOTO must have a label; otherwise, it cannot be transferred to or executed.

```
GOTO i[[,] (s[,s]...)]
```

i Integer variable name, assigned by previous ASSIGN statement.

s Statement label of an executable statement in the same program unit. A given statement label can appear more than once in this statement. CFT77 does not restrict *i* to values shown in this 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:

(1)
```
ASSIGN 76 TO LAB
...
GOTO LAB
```

(2)
```
ASSIGN 999 TO KFIN
...
GOTO KFIN (997,997,999)
```

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

6.4.4 ASSIGN STATEMENT

An ASSIGN statement assigns a statement label to an integer variable and is the only way to define a variable with a statement label. See examples immediately preceding.

ASSIGN *s* TO *i*

s Statement label for an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

i Integer variable name

A variable defined with the label of an executable statement should be referenced only in an assigned GO TO statement or as a format identifier in an I/O statement, and only in the same program unit. While so defined, the variable should not be referenced for any other purpose; if it is referenced, the results are unpredictable.

A variable representing a statement label should be defined only by the ASSIGN statement. Any code like the following is bad practice:

```
      ASSIGN 10 TO I
      ASSIGN 20 TO J
      I=J                    ! Not recommended
```

6.5 SUSPENDING AND HALTING EXECUTION

The remaining control statements are used to show the end of code, or to suspend or terminate execution.

6.5.1 STOP STATEMENT

A STOP statement terminates execution of a main program or procedure subprogram. A program unit can have more than one STOP statement, which can appear anywhere that an executable statement can appear. That is, STOP ends execution, whereas END is the last source statement of a program unit.

STOP [*id*]

id — Identifier of the STOP statement for use in messages; has no effect on the executable program. *id* can be an unsigned integer constant of up to 8 digits, a character constant of up to 8 characters, or the name of a character variable, array element, or function containing or providing 8 characters.

The ANSI Fortran Standard limits noncharacter *id* to 5 digits, sets no limit on the length of character constants, and does not permit *id* to be the name of a character variable, an array element, or a function. The Standard does not specify a use for the identifier.

6.5.2 END STATEMENT

The last source statement in each program unit must be an END statement. In a subprogram it has the effect of a RETURN statement; in a main program it has the effect of a STOP statement. A single END statement can appear in the same program unit with one or more STOP statements and one or more RETURN statements.

If an initial line contains only the characters END in that order, the line cannot have a continuation line. This form of initial line is a *terminal line*. Example:

```
      Legal:     END=1.2       ! (but not an END statement)

      Illegal:   END
                +  =1.2
```

Embedded comments can be included on an END statement when preceded by an exclamation point.

6.5.3 PAUSE STATEMENT

A PAUSE statement suspends or terminates a main program or procedure subprogram. An installation parameter determines whether execution can be resumed or is unconditionally terminated.

PAUSE [*id*]

id — Identifier of the PAUSE statement for use in messages; has no effect on the executable program. *id* can be an unsigned integer constant of up to 8 digits, a character constant of up to 8 characters, or the name of a character variable, array element, or function containing or providing 8 characters.

The ANSI Fortran Standard does not provide for the option of resuming or terminating execution.

The ANSI Fortran Standard limits noncharacter *id* to 5 digits, sets no limit on the length of character constants, and does not permit *id* to be the name of a character variable, array element, or function.

I/O: OVERVIEW, TERMS, STATEMENTS 7

7.1 I/O TUTORIAL

This subsection is not reference material but is intended to show how typical input and output are performed. *Input* statements transfer data from a file to program memory. This process is called *reading*. *Output* statements transfer data from program memory to a file. This process is called *writing*. The CFT77 statements for I/O operations are summarized in table 7-1.

Each I/O statement must specify the file to be read or written, the format of the data, and the items to be transferred. Both the file and format can be specified by asterisks, which indicate defaults and allow the simplest coding. Example:

```
REAL ARR(10)
READ (*,*) I, R, ARR
WRITE (*,*) I, R, ARR
```

In the above READ and WRITE statements, the first asterisk indicates a default file to be used for the data transfer. The second asterisk indicates the default method of formatting, called *list-directed*. These statements transfer the values of variables I and R and array ARR.

Under UNICOS, the default files are **stdin** and **stdout**, which are linked to your terminal. Under COS the default datasets are $IN and $OUT, which are typically used for transfers with your front-end computer, allowing data to be transferred in with your source code and out with output listings. Default files are discussed in 7.5.1.

7.1.1 USING NONDEFAULT FILES

To read or write a nondefault file in standard Fortran, you should first *open* it with an OPEN statement. This statement assigns the file a *unit number*, which you choose (in the range 1-4 or 7-99). After the file is opened, I/O statements identify it only by this number. A CRAY X-MP extension also allows using the file's name instead of a number, without any need to open the file.

Most aspects of file management are handled by the operating system; for example, transferring a file to or from the front end, or (under COS) saving a file so that it is not deleted. The following examples show the

Fortran and operating system actions that are taken for typical needs. For more about units, see 7.5; OPEN statement, 7.8; file names, 7.3.2.

7.1.1.1 Using a data file from the front end

To use a front-end file, such as DATAFL, as input to your program, do the following. (The READ and WRITE statements shown use the asterisk format, assuming the file is suitable for list-directed reading.)

- In the Fortran code, do either of the following.
  - Standard Fortran: Choosing unit 4 for the file, insert OPEN(4,FILE='DATAFL',STATUS='OLD') in the program. You can then read the values of variables R and I with READ(4,*)R,I .
  - Using the CRAY X-MP extension allowing direct use of a file name: You can read the values of variables R and I from DATAFL with READ('DATAFL',*)R,I. The first such use of the file name opens the file; no OPEN statement is needed.
- Operating system actions: To transfer DATAFL from the front end to a Cray system, do one of the following.
  - Under UNICOS, use the **fetch** command if you have a station, or, with TCP/IP, use **rcp** or the **get** command under **ftp**.
  - Under COS, insert in the JCL file, before commands for executing the CFT77 program, the command FETCH,DN=DATAFL.

7.1.1.2 Creating a file to be transferred to the front end

The following actions do the following: create Fortran file PRFILE, write to the file using list-directed I/O, and transfer it to a front-end computer.

- In the Fortran code, do either of the following.
  - Standard Fortran: Choosing unit 7 for the file, insert OPEN(7,FILE='PRFILE',STATUS='NEW') in the program. You can then write variables R and I to PRFILE with WRITE(7,*)R,I.
  - On the CRAY X-MP system, you can write variables R and I to PRFILE with WRITE('PRFILE',*)R,I. The first such use of the file name opens the file; no OPEN statement is needed.

- Operating system actions: To transfer PRFILE to a front-end computer, such as for printing, do one of the following.

  - Under UNICOS, use the **dispose** command if you have a station, or, with TCP/IP, use **rcp** or the **put** command under **ftp**.

  - Under COS, insert in the JCL file, after commands for executing the CFT77 program, the command DISPOSE,DN=PRFILE. See 7.7.4 concerning printing.

7.1.1.3 Creating a file for further processing

If you wish a program, PGM1, to create a UNICOS file or COS dataset so that another program, PGM2, can use it as input, do the following (for Fortran file KEEPER). The read and write operations shown use *unformatted* I/O, discussed in 7.1.2.1, following.

- In the Fortran code, do either of the following.

  - Standard Fortran: You must open KEEPER in both programs. Choosing unit 8, insert OPEN(8,FILE='KEEPER',STATUS='NEW') in PGM1. Use OPEN(8,FILE='KEEPER',STATUS='OLD') in PGM2. PGM1 can write the values of variables R and I to file KEEPER with WRITE(8)R,I ; PGM2 can read them with READ(8)R,I.

  - On the CRAY X-MP system: PGM1 can write variables R and I to KEEPER with WRITE('KEEPER')R,I; PGM2 can read them with READ('KEEPER')R,I. No OPEN statements are needed.

- Operating system actions:

  - UNICOS automatically creates a permanent file from a Fortran file; you do not need to take further action.

  - Under COS, if PGM1 and PGM2 are different jobs, you need to add commands to the jobs' JCL files, in order to save file KEEPER as written by PGM1, and to let PGM2 access it. Following the JCL commands for executing PGM1, insert SAVE,DN=KEEPER. Preceding the JCL commands for executing PGM2, insert ACCESS,DN=KEEPER. These additions are not needed if PGM1 and PGM2 are in the same job.

7.1.2 THE THREE KINDS OF STANDARD I/O

Standard Fortran can transfer data between a program and another entity in three ways: unformatted I/O, list-directed I/O, and formatted I/O. Cray Research provides additional forms of I/O; see section 9 and the Programmer's Library Reference Manual, publication SR-0113.

7.1.2.1 Unformatted I/O

Unformatted I/O (see 8.1) does not convert data from its internal (binary) representation and is not suitable for printing or for use by other vendors' computers. Unformatted I/O is fast and retains the full precision of any numbers used by a program; it is appropriate when a file is needed primarily to be read by another program, as shown in the previous paragraph, Creating a file for further processing. Example of unformatted I/O:

First program:

```
WRITE (9) ARR1,ARR2          ! To unit 9, no format specified
```

Second program:

```
READ (9) ARR1,ARR2           ! Reads same info as above
```

The above statements transfer arrays ARR1 and ARR2. The unit number (9) is the shortest form of a *control information list* (see 7.7.1); if this does not include a format specifier, the transfer is unformatted.

7.1.2.2 List-directed I/O

List-directed I/O (see 8.2) reads or writes data in the form of ASCII characters, and is the default format where formatting is used (indicated by an asterisk for a format specifier). A list-directed file can be transferred to other vendors' systems and can be printed. Each data item is written or read in a manner appropriate to its data type, at full precision. In printed output, items are separated by commas unless a character string is included.

In the following example, the WRITE and PRINT statements give equivalent results, except that they specify different destinations.

```
WRITE (7,*) A,B,C,'   Best: ',BST          ! To unit 7
WRITE (*,*) A,B,C,'   Best: ',BST          ! 1st * is default unit
PRINT *, A,B,C,'   Best: ',BST             ! To the default unit
WRITE ('RESULTS',*) A,B,C,'   Best: ',BST  ! To file RESULTS
```

In the second WRITE statement above, the first asterisk indicates the default unit. All other asterisks are format specifiers, indicating list-directed I/O. The PRINT statement is like WRITE but only uses the default unit. The last WRITE does not require a preceding OPEN statement (CRAY X-MP extension). All four statements above would give the following result:

```
3.316624790355, 2.236067977, 4.   Best: 4.
```

7.1.2.3 Formatted I/O

Formatted I/O (see 8.3), like list-directed I/O, writes and reads data in the form of ASCII characters, but specifies the format of the data. Formatted output can be printed or ported across different computers and operating systems, but this is the slowest kind of I/O. The format is specified as a list of *format descriptors* (see 8.4) enclosed in parentheses. A format can be used by a READ, WRITE, or PRINT statement entered as either the number of a FORMAT statement (see 8.3), or as a character expression (such as a literal string enclosed in single quotes; see examples 2 and 3 following). In the following example, the WRITE and PRINT statements give equivalent results, except that they specify different destinations.

```
      WRITE (*,15) PEOPLE,ALTITUDE,FUEL      ! To default unit, format 15
      PRINT 15, PEOPLE,ALTITUDE,FUEL         ! To default unit, format 15
      WRITE (8,15) PEOPLE,ALTITUDE,FUEL      ! To unit 8, format 15
      WRITE ('AIR',15) PEOPLE,ALTITUDE,FUEL  ! To file AIR; non-ANSI

 15   FORMAT (' Number of passengers:     ',I2/
     &   ' Altitude:',13X,I5,' feet'/
     &   ' Remaining fuel:',6X,F6.1,' gallons')
```

Format 15 above uses the I and F descriptors for integer and real values (see 8.4.8.3, 8.4.8.4), X for skipping spaces (8.4.3.2), and / for starting a new line or record (8.4.4). All four statements shown above would give the following result:

```
 Number of passengers:     8
 Altitude:             12371 feet
 Remaining fuel:        29.4 gallons
```

7.1.3 EXAMPLES OF FORMATTED I/O

The following examples are intended to show aspects of I/O coding but do not represent the fastest kind of I/O. See 7.1.4.

(1) Formatting an array. This example shows the use of an *implied-DO list* (see 7.7.2.3) and a format *repeat specifier* (see 8.3). These allow you to control the formatting of each array element in an ASCII file, to allow a particular presentation of data. (Because implied-DO lists hurt I/O performance, use them only to specify either a subset of an array or an element order that differs from the array's storage sequence.)

```
      OPEN(8,FILE='KEEPFL',STATUS='NEW')
      REAL ARRAY1(10)
      WRITE(8,50) (ARRAY1(L),L=3,7)     ! Only elements 3 to 7 are printed
   50 FORMAT(5(3X,F5.2))                ! Insert spaces between elements
```

The above implied-DO list specifies elements 3 through 7 of array ARRAY1; the number of elements, 5, agrees with the repeat specifier in format 50. A READ statement would be coded the same way, except that the OPEN statement would show STATUS='OLD'. For a write operation, format 50 could also use the string '   ' in place of 3X, but a read operation cannot include a literal string.

In a read operation from an ASCII file, an array is read into a Fortran program as a group of numeric values, not as characters; that is, the write process is completely reversed, except that the data might have fewer digits of precision than it had in the program that originally wrote the file.

(2) Shorthand for example 1

```
            WRITE ('KEEPFL','(5(3X,F5.2))') (ARRAY1(L),L=3,7)
                    ------   ------------
                       |           |
Replaces OPEN statement           Replaces FORMAT statement
```

For many purposes, the above statement is equivalent to the statements shown in example 1. If a format is needed by only one transfer statement, it can be put directly in the statement as a character constant in single quotes (apostrophes). Similarly, the file name can be used instead of a number (CRAY X-MP extension).

(3) Formatting a two-dimensional array. To see how a two-dimensional array is formatted, consider an array that is defined as follows:

```
      INTEGER IARR(3,4)
      DO 10, J=1,3
        DO 20, K=1,4
          IARR(J,K)=10*J+K      ! Element values echo subscript values
20      CONTINUE
10    CONTINUE
```

When an array name appears in a WRITE statement without subscripts and with no implied-DO list, the array's elements are printed in the order of their storage sequence, as shown in figure 4-2. Specified in this way, array IARR could be formatted as follows:

```
      WRITE(*,90) IARR
90    FORMAT (/4(25X,3(I2,2X)/))
```

The above repeat specifiers 4 and 3 correspond to the dimensions of the array. Each / symbol starts a new line, and *n*X causes *n* spaces to be skipped. The resulting print-out is as follows:

```
11  21  31
12  22  32
13  23  33
14  24  34
```

Notice that the above arrangement of elements is reversed from the conventional arrangement of rows and columns used in mathematics. For example, element IARR(2,1) is the second element of the first row. To print an array according to mathematical convention, a WRITE statement needs a nested implied-DO list (which decreases I/O speed). The write operation is then coded as follows:

```
      WRITE(*,91)  ((IARR(J,K),K=1,4),J=1,3)
91    FORMAT (/3(25X,4(I2,2X)/))
```

To agree with the element order specified by the implied-DO list, the above repeat specifiers 3 and 4 are reversed from their positions in the previous format. The resulting print-out is as follows:

```
11  12  13  14
21  22  23  24
31  32  33  34
```

7.1.4 INCREASING I/O PERFORMANCE

You can speed up I/O operations by minimizing the amount of processing required. Following are some basic ways to accomplish this:

- Unformatted I/O is considerably faster than formatted I/O (by a factor of 130 for transferring a 1000-element array).
- Naming a whole array with no implied-DO list is faster than using an implied-DO (by a factor of 4 for a 1000-element array).
- BUFFER IN and BUFFER OUT (see 9.1) increase performance by allowing other processing to occur during a transfer. These statements can be further speeded up by the use of the pure data/unblocked file structure (see 9.2.1 and 7.3.1).

More advanced methods of I/O are introduced in 9.2.

Table 7-1. CFT77 Input/Output Statements

Statement	Description and Subsection Reference
READ	Transfers data from a file to the program (7.7)
WRITE	Writes to a file using a control information list (7.7)
PRINT	Writes to default output, specifying only a format (7.7)
FORMAT	Formats data transferred between program and file (8.3)
OPEN	Initializes a file for I/O operations (7.8)
CLOSE	Returns a file to the operating system (7.9)
INQUIRE	Returns information about a file's properties (7.10)
BACKSPACE	Repositions file before the preceding record (7.11.1.2)
ENDFILE	Writes end-of-file at file's current position (7.11.1.2)
REWIND	Repositions file to the beginning of data (7.11.2.3)
NAMELIST†	Enables data transfer between the program and a file containing lists of variables with assigned values (9.3)
BUFFER IN†	Initiates transfer of data from a file to the program; allows concurrent execution of later statements (9.1)
BUFFER OUT†	Initiates transfer of data from the program to a file; allows concurrent execution of later statements (9.1)

† CFT77 extension

7.2 INPUT/OUTPUT RECORDS

A *record* is the smallest entity that can be read or written by a Fortran I/O statement. A record is a sequence of values or characters, typically a line. For printed output, each print line is a record. A record may or may not correspond to a physical entity.

Records can be of the following types:

- 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 the record is written. The length also depends on characteristics of the peripheral device (for example, a printer or terminal) serving as the origin or ultimate destination of the data. Formatted records can be read or written by formatted I/O statements, or prepared by means other than Fortran.

  Unformatted and buffered I/O statements can also read and write formatted records, but in a manner ignoring their formatted characteristics. Because of record blocking, reading formatted records with unformatted I/O statements may not be practical. The structure of COS blocked records is described in the COS Reference Manual, publication SR-0011.

- An *unformatted* record consists of a sequence of character and/or noncharacter data. The length of an unformatted record is measured in storage units (words) unless the record contains character data items. In that case, each character entity of length *len* takes ((*len*-1)/8)+1 words. Unformatted records can be read or written by unformatted and buffered I/O statements. Unformatted records cannot be read or written with formatted I/O.

- An *end-of-file* (endfile) record occurs as the last record of a blocked file. An endfile record can be written at the end of a file by an ENDFILE statement (see 7.11.2.2).

- An *end-of-data* (EOD) record occurs as the last record of a UNICOS blocked file or COS dataset. It cannot be explicitly written by any Fortran statement except a CLOSE statement (see 7.9). In COS, the utility function EODW can be called from a CFT77 program to write an EOD record.

The ANSI Fortran Standard does not provide for end-of-data records.

7.3 INPUT/OUTPUT FILES AND DATASETS

A *file* is a sequence of records. A CFT77 data file can contain formatted records, unformatted records, or (only under COS) a combination of both.

I/O statements perform operations on Fortran files. A Fortran file can be either internal or external:

- An *internal file* (see 7.4) is internal to the program and ceases to exist when the program terminates. Internal files are not associated with operating system files.

- An *external file* is a file that is associated with a UNICOS file or COS dataset. UNICOS automatically makes a file permanent after it is created by a Fortran program, but COS requires a SAVE command in the program's JCL file to make the file permanent. (Note: a scratch file is considered external but is not associated with a UNICOS file or COS dataset.) To create Fortran files and assign unit numbers, do the following:

  - Standard Fortran: An external file has a unit number (see 7.5). The OPEN statement (see 7.8) can create a file and associate it with a unit number, or the file can already exist and be preassociated with a unit.

  - CRAY X-MP extension: An external file that does not already exist is created when its name (within single quotes) is used in place of a unit number in a data transfer statement. The file can be accessed in the same way in subsequent transfer statements but has no unit number.

See 7.9 concerning explicit and implicit closing of files.

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

7.3.1 FILE STRUCTURES

CFT77 uses file structures with the following characteristics:

- The *blocked* file structure is used for UNICOS sequential unformatted I/O and for all COS I/O except as noted in the following paragraph concerning the unblocked structure.

  Each block (512 words) begins with a block control word (BCW), and each record is terminated by an end-of-record word (EOR). EOR is one kind of record control word (RCW); the others are end-of-file (EOF) and end-of-data (EOD). A file ends with the following records, in this order: EOR; an empty record called an *endfile;* EOF; EOD. Each record begins on a word boundary.

- The UNICOS *text* file structure consists of 8-bit ASCII characters; each record is terminated by a **\n** character. This file structure is used for all UNICOS formatted I/O, either sequential or direct access.

- The UNICOS *pure data* file structure and COS *unblocked* file structure have no record boundaries. You can get faster performance with this file structure, but you must know the data length for each read operation. This structure can be used in the following ways:

  - Unformatted direct access I/O: system- and library-buffered, synchronous, with buffer size control. Each item in the *iolist* must be an array name without subscripts, for an array whose dimension sizes are multiples of 512.

  - BUFFER IN/BUFFER OUT (see 9.1) with pure-data or unblocked structure is asynchronous and only system-buffered. The LENGTH function will not work.

  - Random I/O (see 9.2) generally uses this structure. Some subroutines add their own structure.

  The pure-data/unblocked structure must be specified to the operating system, as shown in subsection 9.2.1.

The structure of blocked files and datasets is further described under BLOCKED in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014, and in the COS Reference Manual, publication SR-0011.

UNICOS does not have datasets or any other multiple file entity. To use more than one file, a program must separately associate each file with a unit number, in an OPEN statement or by other means (see 7.8 and 7.8.1).

7.3.2 FILE IDENTIFIER

A file identifier specifies a file. It can be one of the following:

- A character expression is the name of an external file, and can appear with the FILE= specifier in an OPEN, CLOSE, or INQUIRE statement. UNICOS allows 14 characters for any file name (including a directory name) and 128 characters for the complete path name. COS allows 7 characters for a dataset name. If the expression is a literal name, it is enclosed in single quotes (apostrophes). Examples:

  ```
  OPEN (UNIT=8, FILE='BEAUTY')
  OPEN (UNIT=9, FILE=FILENM)
  OPEN (UNIT=10, FILE=FNAME//FNUM)
  ```

  FILENM above is a character variable whose value is the file name. The character expression FNAME//FNUM is evaluated, and its result is the file name.

- A character constant of up to seven characters (enclosed in single quotes) is the name of an external file, and can appear in place of a unit number in a READ, WRITE, BACKSPACE, ENDFILE, or REWIND statement (CRAY X-MP extension; see 7.7.1). Example:

  ```
  READ ('INFILE',100) A
  WRITE ('OUTFILE',200) A
  ```

- The name of an internal file can appear in a READ, WRITE, or PRINT statement. It is the name of a character variable or other character entity (which is not enclosed in single quotes). See 7.4. Example:

  ```
  WRITE (FILTER,300) ARRAYM
  ```

- An integer variable or array element containing Hollerith data constitutes an external file name that can be used in the same contexts allowed for a character expression, with the same size limits.

The ANSI Fortran Standard does not provide for file identifiers to be used in place of unit identifiers.

7.3.3 COS DATASET

Under COS, a *dataset* is a sequence of files identified by a single name and having the same unit number. Whether a file's unit number is preassociated or assigned by the OPEN statement, all files in the same dataset have the same unit number. The use of datasets and related JCL commands are discussed in 1.3 and, in section 7, under the heading Using Nondefault files. Datasets are further described in the COS Reference Manual, publication SR-0011.

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

7.3.3.1 Example: writing two files

The following program, executing under COS, writes two files within a dataset. It creates a CFT77 file named CIRCLES; reads an input file containing values for the diameters of circles; and writes (to file CIRCLES) the diameters, an end-of-file mark, and the circumferences.

```
      PROGRAM TWOWRITE
      PARAMETER(PI=3.14)
      DIMENSION CIRCUM(10)
      OPEN(UNIT=1,FILE='CIRCLES',STATUS='NEW')
      DO 5 I=1,10
        READ(*,*,END=10) DIAM          ! Read diameter from input file
        WRITE(1,*) DIAM                ! Write diameter to file CIRCLES
        CIRCUM(I)=DIAM * PI
    5 CONTINUE
   10 ENDFILE 1                        ! At end of read, write EOF marker
      WRITE(1,*)(CIRCUM(J),J=1,I-1)    ! Implied-DO writes to file CIRCLES
      CLOSE(1)
      END
/EOF
6.15
72.54
18.42
/EOF
```

CFT77 associates file CIRCLES with a COS dataset of the same name. Although CIRCLES is treated as one file by the above WRITE statements, at program termination it is (in COS terms) a dataset containing two files: one containing the diameters and the other containing the circumferences. The division is established by the ENDFILE statement.

Defaults can be used in the preceding example: if the OPEN statement is omitted and the WRITE statements use * as the unit identifier, the data is written to dataset $OUT.

7.3.3.2 Example: reading two files

Files within datasets do not have individual identifiers but are accessed sequentially with the READ statement (see 7.7), using its END parameter to detect file boundaries within the dataset. The following program reads and prints the two files that were written in the previous example.

```
      PROGRAM TWOREADS
      REAL CIRCUM(10),DIAM(10)
      OPEN(UNIT=1,FILE='CIRCLES',STATUS='OLD')
      READ(1,*,END=10)(DIAM(I),I=1,10)
   10 READ(1,*,END=30)(CIRCUM(J),J=1,10)
   30 PRINT*,('DIAMETER = ',DIAM(K),'CIRCUMFERENCE = ',CIRCUM(K),K=1,10)
      END
```

The second READ statement above reads the second file in dataset CIRCLES because the first READ statement executes until the first end-of-file marker is reached. As in the preceding example, the OPEN statement treats the dataset as a single CFT77 file.

7.4 INTERNAL RECORDS AND FILES

An *internal* file allows conversion of data within a program, using the full range of format descriptors described in section 8. An internal file is in fact a character variable, or other character entity, that is written to as if it were a file. In I/O statements, the name of the character variable (that is, the name of the internal file) is used in place of a unit identifier. Internal files are useful for changing the form of input data before using it.

An internal file is a character variable, character array element, character array, or character substring. A record of an internal file is a character variable, character array element, or character substring. If the internal file is a character variable, character array element, or character substring, it consists of a single record with the same length as the file. If the internal file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the internal file. The ordering of the file records is the same as the ordering of elements in the array. Every record of the file has the same length as an element in the array.

The contents of a record of the internal file are defined by writing the record with an output statement, or by modifying it as a character variable, such as with a character assignment statement or an input statement. 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.

An internal file is always positioned at the beginning of the first record before data transfer. Reading and writing records is done only by sequential access formatted I/O statements not specifying list-directed formatting.

Example:

```
      INTEGER POSN
      CHARACTER *10 TEMP              ! Variable TEMP, to be internal file
      ...
      READ(6,1)TEMP                   ! Read record from device 6 to TEMP
    1 FORMAT(A10)
      IF (INDEX(TEMP,'$').NE.0) THEN  ! Replace dollar
         POSN = INDEX(TEMP,'$')       ! sign with
         TEMP(POSN:POSN) = ' '        ! a blank
      ENDIF
      READ(TEMP,2) VALUE              ! Read result into real variable
    2 FORMAT(F10.2)
```

The above statements read numbers from an input file that have been entered with a dollar sign preceding them. The statements place an input value in an internal file (that is, character variable TEMP), remove the dollar sign, then transfer the value to a type REAL variable.

7.5 UNITS

A *unit* is a means of referring to a file or dataset; it equates a file used by a Fortran program and a file known to the operating system. At a given time, a set of units *exists* for an executable program; these are the units that can be connected in OPEN statements. See 7.9 concerning explicit and implicit closing of files.

A *unit identifier* is a means of referring to a file and is an integer constant or expression in the range 0-101 (UNICOS) or 1-102 (COS), or the character *. Permissible kinds of unit identifiers are described under UNIT= in 7.7.1.

A file identifier can be used in place of a unit identifier; this can be for an internal file (see 7.4) or (only on the CRAY X-MP system) an external file. If the file does not exist, it is created and the operation proceeds. See 7.3.2 and 7.7.1.

7.5.1 DEFAULT UNITS

Two kinds of defaults (that is, preconnections) are used for unit numbers:

- The asterisk is used only in READ and WRITE statements as a default; in addition, the PRINT statement always uses the default output unit, although no asterisk appears. The default unit is used only for formatted, sequential transfers; it is always unit 100 for input and unit 101 for output.

  Under UNICOS, 100 and 101 are preconnected to files **stdin** and **stdout**, respectively; under COS, to datasets $IN and $OUT. In addition, unit 102 under COS is preconnected to $PUNCH (see E.4). These assignments cannot be changed, and the numbers cannot appear in an OPEN statement. (Therefore they do not exist, either as defined at 7.5 or as reported by an INQUIRE statement.)

- Units 5, 6, and 0 are preconnected to UNICOS files **stdin, stdout,** and **stderr**, respectively; units 5 and 6 are preconnected to COS datasets $IN and $OUT. These unit numbers are not equivalent to the asterisk. They can be reassigned, but the effects of such reassignment depend on the machine and operating system in use.

The ANSI Fortran Standard does not include the following:

- A maximum value for the external unit identifier
- Provision for the definition and preconnection of unit identifiers 100, 101, 102, 5 and 6
- The use of an external file name as a unit identifier

7.5.2 REDIRECTION TO AND FROM DEFAULT FILES

If you are porting a program from a system whose default files are used differently than the Cray default files available to you, you can take operating system actions to copy between the default files and other files, so that you do not need to change your program. For example, you might want the statements READ(*,10)... or WRITE(*,20)... to access permanent Cray-resident files rather than files that are transferred to or from your terminal or front end. This might be needed for porting from another vendor's computer or for porting between COS and UNICOS. It is done as follows:

- Under UNICOS: To redirect file **indata** to default file **stdin**, and then default file **stdout** to file **outdata**, enter the **a.out** command as follows:

```
a.out < indata > outdata
```

- Under COS: To copy Cray-resident dataset INDS to default dataset $IN, and the output data file of default dataset $OUT to Cray-resident dataset OUTDS, alter your JCL as follows:

```
...
REWIND,DN=$OUT.
SAVE,DN=$OUT,PDN=MYLOG,UQ.
CFT77.
RELEASE,DN=$IN.
ACCESS,DN=$IN,PDN=INDS.
SEGLDR,GO.
SKIPF,DN=$OUT,O=MYOUT,NF=1.
SAVE,DN=MYOUT.
```

  The above JCL assumes that your source program is contained in the next file within $IN. If your program is in a nondefault file, you need an additional FETCH or ACCESS statement along with the I=*name* parameter on the CFT77 statement, as shown in 1.2.3.

7.6 I/O FORMATS

A format specifies the form of input or output data. Formats are identified in most I/O statements, as described in 8.3. A format identifier must be one of the following:

- A FORMAT statement label appearing in the same program unit as the format identifier (see 8.3.1)
- An integer variable name that has been assigned the value of a FORMAT statement label, in an ASSIGN statement. The variable name cannot also appear as a dummy argument in the same program unit.
- An asterisk, specifying list-directed formatting (see 8.2)
- A character expression whose value is a format specification. For instance, a format can be entered directly in a data transfer statement as a literal string enclosed in single quotes (apostrophes). Examples:

```
WRITE(*,'(F7.3,4X,I3)')C,J
WRITE(*,FMTARR(M))P,Q,R,S
READ(11,SPECA//SPECB)A,B
```

  The above statements use different kinds of character expressions as format specifiers: a literal string, a character array element, and an expression using the concatenation operator. The values of these expressions are used for formatting.

- A character array name, representing a single format. The elements of the array are combined in ascending order into a single string. For character array FA with three elements, the following specifications are equivalent:

```
WRITE(*,FA)...
WRITE(*,FA(1)//FA(2)//FA(3))...
```

7.7 READ, WRITE, AND PRINT STATEMENTS

Data transfer statements cause the transfer of data between files and other entities, such as printers or disk drives.

The READ statement is the input statement; it causes values to be transferred from an external or internal file to the entities specified in the input list, if present.

The WRITE and PRINT statements are output statements; they cause values to be transferred from the entities specified in the output list, if present, to an external or internal file. As shown in the following formats, PRINT does not allow the use of a control information list.

READ (*cilist*)[*iolist*]
READ *f* [,*iolist*]
WRITE (*cilist*)[*iolist*]
WRITE *f* [,*iolist*] (*non-ANSI*)
PRINT *f* [,*iolist*]

cilist Control information list; includes a reference to the source or destination of the data to be transferred and an optional format identifier. Other *cilist* entries are shown in 7.7.1.

f Format identifier

iolist I/O list specifying the data to be transferred

Examples, as shown in subsection 7.1.2:

```
WRITE (*,*) I, R, ARR
READ(4,*)R,I
READ('DATAFL',*)R,I
WRITE(8,50) (ARRAY1(L),L=3,7)      ! Writing with implied DO
```

The ANSI Fortran Standard does not provide for the WRITE *f* [,*iolist*] format.

7.7.1 CONTROL INFORMATION LIST

The *control information list* (*cilist*) can be used in data transfer statements to specify the source or destination of data, the data format (if any), a statement where execution continues at an error or end of data, a record number for direct access I/O, or a variable to receive status information.

[UNIT=]*id*[,[FMT=]*f*][,END=*sn*][,REC=*rn*][,ERR=*s*][,IOSTAT=*ios*]

[UNIT=]*id*

Unit identifier or file identifier; one or the other must be specified, but not both. If the UNIT= keyword is omitted, parameters are positional, and the file identifier must appear first in the list.

A unit identifier (see 7.5) corresponds to an external file; it is either: an integer constant or expression in the range 0-101 (UNICOS) or 1-102 (COS); or (in a READ or WRITE statement) the character * for a default value (see 7.5.1). Nondefault unit numbers 1-4 and 7-99 are connected with the OPEN statement (see 7.8).

A file identifier for a character variable or array (that is, a name not enclosed in single quotes) specifies an internal file (see 7.4). A file identifier of up to seven characters in single quotes refers to an external file (CRAY X-MP extension). If an external file specified in place of the unit does not already exist, it is created; that is, it is not required to be opened in an OPEN statement. It can be saved or disposed by normal operating system actions but is not given a unit number.

The ANSI Fortran Standard does not provide for an external file identifier to be used for the [UNIT=] parameter, nor for reading or writing to a file that does not exist.

[FMT=]*f* Format identifier (see 7.6, 8.3). This parameter must be present for formatted I/O statements. If *f* is an asterisk, the statement is list-directed and a record identifier cannot be present. If the optional UNIT= keyword is specified with the unit or file identifier, the FMT= keyword must be specified with the format identifier. If both the UNIT= and the FMT= keywords are omitted, *f* must follow the identifier for the [UNIT=] parameter.

END=*sn* End-of-file identifier. *sn* is the number of the statement where execution continues after an EOF on a READ statement has been encountered. An end-of-file identifier must not appear in a WRITE statement or in the same control information list as a record identifier. Under UNICOS, control-D sends an end-of-file identifier from a terminal.

REC=*rn* Record identifier. *rn* must be an integer expression with a positive value. A record identifier appears only in direct-access I/O statements (see 7.11.1). A statement containing a record identifier cannot contain an end-of-file identifier.

ERR=*s* Error identifier. *s* is the statement label of the statement where control continues after a recoverable error occurs.

IOSTAT=*ios*

Status identifier that becomes defined when an I/O statement is executed. *ios* must be an integer variable or an integer array element. Following are the identifier values and their meanings:

=0 Transfer is complete; no error or end-of-file condition exists.

>0 Error message number; see coded $IOLIB messages in COS Message Manual, publication SR-0039.

<0 End-of-file was encountered; no error condition exists.

Examples of control information lists:

Statement	Comment
READ(10)...	Unit 10, unformatted
WRITE(10,430)...	Unit 10, format 430
WRITE(10,REC=J)...	Unit 10, direct access
READ('FILE1',30)...	External file FILE1, format 30
READ(*,*,END=200)...	Unit 100 (default); list-directed; jump to statement 200 at end of data.
READ(END=100,FMT=20,UNIT=5)...	Order is optional with keywords
WRITE(98,'(6E11.4)',ERR=75)...	Format is a character expression
READ(J,ARRAYF,ERR=10,END=25)...	Unit id is the value of J; format is a character array.

7.7.2 I/O LIST

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

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

7.7.2.1 Input list items

Only input list items can appear in an input statement. An *input list item* must be one of the following:

- Variable name
- Array element name
- Array name
- Character substring name

7.7.2.2 Output list items

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

- Variable name
- Array element name
- Array name
- Character substring name
- An expression, with the following exceptions: an array syntax expression; or a character expression in which one operand's length is specified as (*).

Examples of input and output lists:

```
(1)   READ(23)X,Y                    ! Variable names

(2)   WRITE(23)A(1),A(4),X(2)        ! Array element names


(3)   DIMENSION Z(64)
       ...
      READ(23)Z                      ! Array name


(4)   CHARACTER*10 WORD
       ...
      READ(23)WORD(2:3)              ! Character substring


(5)   WRITE(23)A+B                   ! Expression
```

7.7.2.3 Implied DO list

An implied-DO list allows a statement to operate on a list of entities systematically as in a DO-loop. The following format represents an I/O list or one item in an I/O list.

(*dlist*, *dvar* = *init*,*lim*[,*incr*])

dlist An I/O list. List items are separated by commas.

dvar Name of an integer variable called the *implied-DO variable*. dvar *is initially defined with* init *and is incremented by* incr *on each iteration of the loop. This variable has the normal range for a variable: the whole program unit.*

init, *lim*, and *incr*

Initial value, limit, and increment of the implied-DO variable; must be expressions containing only integer constants, the names of integer constants, and implied-DO variables of other implied-DO lists containing this implied-DO list within their ranges. *incr* must be nonzero; it defaults to 1.

The range of an implied-DO list is the list *dlist*. The trip count and values of the implied-DO variable *dvar* are established as for a DO-loop except that the trip count must be positive. Interpretation of an implied-DO list 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. When an implied-DO list appears within another implied-DO list, the inner list is iterated through its full trip count for each single iteration of the outer list. When the values of *dvar* and of the trip count are established, *dvar*, *init*, *lim*, and *incr* can be redefined with no effect on the loop control process.

A DO variable in an implied-DO list becomes defined at the beginning of processing the implied-DO list as an I/O list item. 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 (also see example 3in 7.1.3):

```
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)
```

7.7.3 DATA TRANSFER OPERATION

When a data transfer I/O statement (READ, WRITE, or PRINT) is executed, the following operations are performed in the order specified.

1. The direction of data transfer is determined (input for READ, output for WRITE and PRINT).
2. The unit involved in the transfer is identified (see 7.5 and 7.7.2 under UNIT=).
3. The format (if specified) is established (see 7.6 and 7.7.2 under FMT=).
4. Data is transferred between the external or internal file and the entities specified by the I/O list (if any).
5. The status identifier (if specified) is defined.

7.7.3.1 Transferring data

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

All values needed to determine entities specified by an I/O list item are determined at the beginning of the processing of that item. For example, the following statements cause a value to be read into N(3).

```
N(1)=3
READ(8)N(N(1))
```

All values are transmitted to or from the entities specified by a list item before the processing of any succeeding list item. For example, the following statement causes two values to be read.

```
READ(3)N,A(N)
```

The first value read is assigned to N, and the second is assigned to A(N), where the new value of N is used as the subscript.

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

7.7.3.2 Unformatted data transfer

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

On input, the file should be positioned so the record read is an unformatted record or an endfile record. (Under COS, CFT77 allows formatted and unformatted records on the same file or dataset (non-ANSI). This is not allowed in UNICOS.) The number of values required by the input list must be less than or equal to the number of values in the record and must not require more values than the record contains.

7.7.3.3 Formatted data transfer

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

On input, the record read should be a formatted record or an endfile record. (Under COS, CFT77 allows formatted and unformatted records on the same file or dataset (non-ANSI). This is not allowed in UNICOS.)

The I/O list and format specification must not specify more than 152 characters. Some formats larger than 133 characters generate warning errors. If the input record length is less than the input list requires, the additional characters are defined as blanks.

The ANSI Fortran Standard does not provide for a maximum number of characters per record, nor for blank padding if the record is less than that required for the input list.

7.7.4 OUTPUT TO A PRINTER

The transfer of formatted record information to certain devices is called *printing*. The first character of a formatted record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines the vertical spacing to occur before printing. The character codes specifying vertical spacing (carriage) control are shown in table 7-2. Under UNICOS, a file with these control characters can be filtered by routine **asa**.

If the record contains no characters, an advance of one line occurs and nothing is printed in that line. A PRINT statement can be used for carriage control without printing any characters.

Table 7-2. Print Control Characters

Character	Vertical Spacing Before Printing
Blank	Advance one line
0	Advance two lines
1	Advance to first line of next page
+	No advance
All other†	Advance one line

† Certain other characters have an effect on carriage control in other operating systems. Refer to the documentation for your front-end processor's operating system if you intend to use a front-end processor to print output files.

7.7.5 ERROR AND END-OF-FILE CONDITIONS

If an error condition occurs during data transfer, the position of the file is indeterminate.

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

If no error condition or EOF condition exists, the file is positioned after the last record read or written.

If an error condition or EOF condition is encountered during a read operation, the read terminates and the entities specified in the I/O list become undefined.

7.7.6 RESTRICTIONS ON INPUT/OUTPUT STATEMENTS

A function must not be referenced in an I/O statement if it causes an I/O statement to be executed.

An I/O statement must not reference a unit or file not having all the properties required for its execution. For example, an attempt to read from a printer will fail.

7.7.7 I/O ERROR RECOVERY

If an irrecoverable error occurs during the execution of an I/O statement, the operating system aborts the current job step. The current job step is aborted even if an error identifier (ERR=*sn*) appears in the I/O statement's control information list. Generally, error conditions detected by code in LIBIO under UNICOS or in $IOLIB under COS are recoverable and return control to the statement indicated by the error identifier; error conditions detected by the operating system are irrecoverable and abort the current job step.

The ANSI Fortran Standard does not distinguish between recoverable and irrecoverable errors.

7.8 OPEN STATEMENT

The OPEN statement establishes an external file for use in a Fortran program. The file is assigned a unit number that you select and specify in the OPEN statement. Subsequent I/O statements use this number to specify the file. (No OPEN statement is needed for the default units specified by an asterisk; these are files **stdin** and **stdout** and datasets $IN and $OUT.) Depending on the status of the file, you can do one of the following with the OPEN statement; examples are for a file named F on unit 8.

- Create a new file (specifying a unit number, which you select): OPEN(8,FILE='F',STATUS='NEW'). Under COS, if this file is to be kept in Cray storage, insert in the JCL file for the Fortran program, SAVE,DN=F. Under UNICOS, the file is automatically made permanent. To transfer the file to a front-end computer, use the **dispose** command under UNICOS; under COS, use DISPOSE,DN=F.
- Allow the Fortran program to access an existing file by assigning it a unit number: OPEN(8,FILE='F',STATUS='OLD'). Under COS, if this file is in Cray storage, insert in the JCL file for the Fortran program, ACCESS,DN=F.
- Create a file that is preconnected to a unit
- Change the characteristics of an existing connection between a file and a unit

Notice that most specifiers in an OPEN statement must be character expressions. This allows the use of a character variable, and it means that a specifier entered literally must be enclosed in single quotes (apostrophes).

```
OPEN (olist)
```

olist An external unit identifier and at most one of each of the other identifiers described in table 7-4

Even if a unit is already connected to a file, an OPEN statement can connect a different file to the same unit; this causes the first file to be disconnected (equivalent to a CLOSE statement with no status identifier). If the FILE= identifier is not included in the OPEN statement, the OPEN statement applies to whatever file is already connected to that unit, if any.

If the file to be connected to the unit does not exist but is the same as the file to which the unit is preconnected, the specifications in the OPEN statement become a part of the connection.

If an OPEN statement specifies a file and unit that are already connected, only the BLANK= identifier can have a value that is different from the current value. The new BLANK= value is then used for that connection. The file position is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

7.8.1 ALTERNATIVES TO THE OPEN STATEMENT

The OPEN statement is used for establishing nondefault files, but other methods are available for this purpose, including the following:

- You can open a file by using its name in place of a unit number in the first data transfer statement that is to access the file. (This is a CRAY X-MP extension; see 7.3, 7.3.2, and 7.7 under UNIT=, and Using Nondefault Files, p. 7-1.) Example:

  ```
  WRITE ('NEWFILE',*) R,I
  ```

- As shown in 7.5.1, the default files can be directed to and from nondefault files, using actions of the operating system.

- You can preconnect a file to a Fortran unit. These actions establish the file's *alias,* an alternative file name that includes the file's Fortran unit number); the file does not need to be opened within the Fortran program. Examples:

  UNICOS: **ln infile fort.9**

  ```
  COS:    ACCESS,DN=INFILE.
          ASSIGN,DN=INFILE,A=FT09.
  ```

  When either of the above actions is taken, a Fortran program can read from file **infile** or dataset INFILE with the Fortran statement READ(9,*)R,I. No OPEN statement is needed. Under UNICOS, the **assign** command can also perform this function, as described in the UNICOS User Commands Reference Manual, publication SR-2011.

Table 7-3. OPEN Specifiers and Their Meanings

Specifier	Type *	Meaning	Input Value
UNIT=*u*†	I	External unit number	Unit number; user-selected
FILE=*fin*††	C	File specifier	Name of file to be connected
STATUS=*sta*	C	Disposition specifier Default: 'UNKNOWN'	'OLD': file must already exist; you must specify file with FILE=*fin*.†† 'NEW': creates a file; you must specify a file name with FILE=*fin*.†† Status becomes OLD. 'SCRATCH': file is deleted by a CLOSE statement or when program terminates. 'UNKNOWN'(default)†††: 'SCRATCH' if file is not specified and unit is not connected; otherwise status is 'OLD'.
IOSTAT=*ios*	i	*ios* name is assigned error status: 0 for no error, or message num.	Symbolic name for error status specifier
ERR=*s*	S	Statement transferred to if error occurs	Fortran statement label
FORM=*fm*¶	C	Formatting specifier Default: 'FORMATTED' for sequential access, 'UNFORMATTED' direct	'FORMATTED', formatted I/O; 'UNFORMATTED', unformatted
ACCESS=*acc*	C	Access specifier Default: 'SEQUENTIAL'	'SEQUENTIAL' is access method; 'DIRECT' is access method.
RECL=*rl*	i	Record length for direct access method; omitted for sequential	Formatted I/O: number of characters per record. Unformatted I/O: 8 times the number of words. Maximum 267 on UNICOS, 152 on COS.
BLANK=*blnk*	C	Blank specifier Default: 'NULL' (also applies to default files)	'NULL': numeric input blanks are ignored; 'ZERO': all nonleading blanks are treated as 0's. Allowed on files opened for formatted I/O only. Overridden by BN and BZ descriptors.

All footnotes are on the following page.

Table 7-4. CLOSE Specifiers and Their Meanings

Specifier	Type *	Meaning	Input Value
UNIT=*u*†	I	External unit number	Unit number
IOSTAT=*ios*	i	Error status is assigned to *ios* name: 0 for no error, or message number.	Symbolic name for error status specifier
STATUS=*sta*	C	Disposition specifier Default, 'KEEP' if OPEN status is 'OLD', 'NEW' or 'UNKNOWN'. Default, 'DELETE' if OPEN status is 'SCRATCH' or if file is memory-resident.	'KEEP': dataset continues to exist after CLOSE stmt executes; do not use 'KEEP' for a dataset with 'SCRATCH' status on an OPEN statement. 'DELETE': the dataset does not exist after execution of the CLOSE statement.
ERR=*s*	S	Statement transferred to if an error occurs	Fortran statement label

* The types for tables 7-3 and 7-4 are as follows:

I Integer constant, variable, or array element
i Integer variable or array element
S Statement label
C Character expression

† UNIT= does not need to be included in the unit specification if *u* is the first item in the list.

†† *fin* is a character expression. Other than names passed as character variables, UNICOS allows 14 characters for any file name and 128 characters for the complete path name. If a file name is passed to a subroutine as a character variable to be used in an OPEN statement, it is limited to 7 characters. COS allows 7 characters for a dataset name. See 7.3.2.

††† UNICOS only: If a file specifier is supplied, the status becomes OLD. If no file specifier is supplied but a file exists by the name fort.*u* where *u* is the unit number, the status also becomes 'OLD'. If such a file does not exist, the status becomes 'NEW'.

¶ COS allows formatted and unformatted records in same file (non-ANSI).

7.9 CLOSE STATEMENT

A CLOSE statement disconnects a particular file from a unit, writes an end-of-data record, and rewinds the file.

```
CLOSE (cllist)
```

cllist An external unit identifier and at most one of each of the other identifiers described in table 7-4

A CLOSE statement can appear in any executable program and need not appear in the same program unit as the OPEN statement that opened the file.

A disconnected file or unit can be reconnected within the same executable program either to the same file or unit, or to a different file or unit, provided the file still exists. Files can be deleted within a program by a CLOSE statement that includes STATUS='DELETE'. Under COS, if the file is memory resident (as established by an ASSIGN JCL command), CLOSE deletes the file regardless of the STATUS identifier.

Under UNICOS, all files are closed at termination of program execution. Under COS, automatic file closing occurs only at the end of an entire job, even if the job includes more than one program; datasets are not made permanent unless they are saved with a SAVE command in the job's JCL. Files are not automatically rewound during program termination.

The ANSI Fortran Standard provides an implicit CLOSE for all files upon normal program termination.

The ANSI Fortran Standard does not provide for automatic deletion of files regardless of the STATUS identifier.

7.10 INQUIRE STATEMENT

An INQUIRE statement returns various kinds of information about a unit or file, such as whether a connection exists, the file name, or the kind of I/O that is currently specified. This information is returned as values assigned to variables that you specify. You can inquire about either a unit or a file.

By unit:

```
INQUIRE (u,islist)
```

By file name:

```
INQUIRE (FILE=fin,islist)
```

u An external unit identifier (see 7.5). The unit specified need not be connected to a file.

fin A character expression that specifies the name of an external file. A file name entered literally must be enclosed in single quotes (apostrophes). The file need not be connected to a unit. Under UNICOS, *fin* can include 31 characters. Under COS, *fin* is limited to seven characters, not counting trailing blanks. Any trailing blanks are discarded.

islist A list of inquiry identifiers that contains at most one of each of the inquiry identifiers described in table 7-5.

islist identifiers are in a form such as EXIST=YES or NAME=FN, where YES and FN are user-named variables of types logical and character, respectively. These variables become defined by the INQUIRE and can be referenced in subsequent statements. Example:

```
LOGICAL AROUND
 ...
INQUIRE(FILE='NAMES',EXIST=AROUND)
IF(.NOT.AROUND)OPEN(UNIT=8,FILE='NAMES',STATUS='NEW')
```

The INQUIRE statement above determines if file NAMES exists; if it does not, the program creates an empty file NAMES.

A variable or array element can be referenced as an identifier only once in an INQUIRE statement.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry identifiers except *ios* become undefined. *ex* and *od* always become defined unless an error condition occurs. Other identifiers are undefined if the unit or file inquired about does not exist; *rcl* and *nr* are undefined if an existing file is sequential access.

Table 7-5. INQUIRE Specifiers and Their Meanings

Specifier	Data Type *	Meaning	Return Value
IOSTAT=*ios*	i	Error status specifier	0 if no error condition exists; error message number if error condition exists.
ERR=*s*	S	Statement label where control is transferred if error condition exists	None
EXIST=*ex*	L	Existence specifier	.TRUE. if unit or file exists; else, .FALSE.
OPENED=*od*	L	Connection specifier	.TRUE. if unit and file are connected; else, .FALSE.
NUMBER=*num*	i	External unit specifier	Unit currently connected; if no unit, *num* is undefined.
NAMED=*nmd*	L	Unit name specifier	True if unit is connected to a file with a name; otherwise false
RECL=*rcl*	i	Record length of unit or file connected for direct access	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*	i	Next record	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*	C	File name	File name associated with the unit if file is named; else, undefined.

* i = Integer variable or array element
L = Logical variable or array element
C = Character variable or array element
S = Statement label

Table 7-5. INQUIRE Specifiers and Their Meanings (continued)

Specifier	Data Type *	Meaning	Return Value
ACCESS=*acc*	C	Access specifier	'SEQUENTIAL' is access method; 'DIRECT' is access method.
SEQUENTIAL=*seq*	C	Sequential as possible access method	'YES' if sequential is allowed; 'NO' if sequential is not allowed; 'UNKNOWN' if unable to determine.
DIRECT=*dir*	C	Direct as possible access method	'YES' if direct is allowed; 'NO' if direct is not allowed; 'UNKNOWN' if unable to determine.
FORM=*fm*†	C	Format specifier	'FORMATTED' if file is connected for formatted I/O; 'UNFORMATTED' if file is connected for unformatted I/O.
FORMATTED=*fmt*†	C	Formatted as a possible allowed form	'YES' if formatted is allowed; 'NO' if formatted is not allowed; 'UNKNOWN' if unable to determine.
UNFORMATTED=*unf*†	C	Unformatted as a possible allowed form	'YES' if unformatted is allowed; 'NO' if unformatted not allowed; 'UNKNOWN' if unable to determine.
BLANK=*blnk*†	C	Blank control specifier	'NULL' if null blank control is in effect; 'ZERO' if zero blank control is in effect. Blank control applies only to formatted records.

† COS allows formatted and unformatted records in the same file (non-ANSI).

* i = Integer variable or array element
L = Logical variable or array element
C = Character variable or array element
S = Statement label

7.11 DIRECT AND SEQUENTIAL FILE ACCESS

External files can be accessed in two standard ways: with *sequential access* (the default), records are stored in the order in which they are written, and read back out in the same order; with *direct access*, records can be read or written in any order, and are assigned numbers. *Random* access is a Cray extension of direct access, described in 9.2.

A file can be created to support sequential access, direct access, or both, using the ACCESS specifier in the OPEN statement. If a file supports both methods of access, the first record accessed by sequential access is the record numbered 1 for direct access, the second record accessed by sequential access is the record numbered 2 for direct access, and so on.

While a file is connected for one kind of access (sequential or direct), only I/O statements using that kind of access may be used with the file.

7.11.1 DIRECT FILE ACCESS

In *direct access* operations, records can be read or written in any order; each record is specified by its number, and all records in a file have the same length. To use direct access I/O, do the following:

- In the OPEN statement that opens a direct access file, specify ACCESS='DIRECT' and RECL=*len*, where *len* is the record length. The maximum value for *len* is 267 characters under UNICOS and 152 under COS.

- In subsequent statements that access the direct access file, specify the record number with the REC=*r* specifier, where *r* is a positive integer that must be specified when the record is written. Examples:

```
WRITE(10,REC=J)...
WRITE(11,14,REC=52)...
READ(12,REC=K)...
```

Example:

```
      PROGRAM DIRCTIO
      OPEN(UNIT=50,FILE='DAOUT',FORM='FORMATTED',ACCESS='DIRECT',
     &     RECL=15)
      DO 10, I=1,5
        WRITE(50,900,REC=I)I
 10   CONTINUE
900   FORMAT(I2)
      A=SQRT(2.0)
      WRITE(50,901,REC=3)A
901   FORMAT(G10.4)
      END
```

The above DO loop 10 writes the first five records in file DAOUT with their respective record numbers. The final WRITE statement replaces, in record 3, the value 3 with the value of A. The resulting file has the following content:

```
1
2
1.414
4
5
```

Once established, a record's number cannot be changed. When an input or output operation is performed on a file, the file is positioned at the beginning of the record specified by the next higher record number, which becomes the current record. A record can be overwritten but not deleted.

Records must not be read or written with list-directed or NAMELIST formatting. Multifile COS datasets cannot be direct access.

Random access I/O, described in 9.2, allows records of variable length, and automatic handling of record numbers, using nonstandard library routines in place of Fortran data transfer statements.

Word-addressable I/O can also be performed using standard direct-access I/O statements, if the file being accessed is in the pure data/unblocked structure. I/O operations are then automatically carried out by the word-addressable routines. Files created in this way cannot be used for blocked direct access or sequential I/O. The pure data/unblocked file structure is specified to the operating system as shown in 9.2.1; file structures are discussed in 7.3.1; word-addressable I/O is introduced in 9.2.

7.11.2 SEQUENTIAL FILE ACCESS

Sequential access operations are based on the sequential storage of records within files. The order of the records is the order in which they are written.

When a sequential input operation is performed on a file, the file is positioned at the beginning of the next record, which becomes the current record. When an output operation is performed on a file, a new record is created, becoming the last record of the file. The position of an internal file is always at the beginning of the character variable, array, array element, or substring referenced by the I/O operation.

The BACKSPACE, ENDFILE, and REWIND statements perform operations on sequential files. The BACKSPACE statement positions a file at the beginning of the previous record from the current record; the ENDFILE statement writes an end-of-file (EOF) mark at the file's current position; and the REWIND statement positions the file to the first record of the file. The formats of these statements are as follows:

```
BACKSPACE id

ENDFILE   id

REWIND    id
```

id One of the following:

- Unit number
- File identifier; character expression such as a literal string in single quotes.
- A list enclosed in parentheses, containing the following set of identifiers:

  [UNIT=]*u*
  IOSTAT=*ios*
  ERR=*s*

The preceding list must contain a single external unit identifier or file identifier and can contain at most one of each of the other identifiers. See the UNIT, IOSTAT, and ERR identifiers described for the OPEN and CLOSE statement in tables 7-3 and 7-4, respectively.

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

BACKSPACE, ENDFILE, and REWIND operations on internal files are not allowed.

The ANSI Fortran Standard does not provide for positioning of an unconnected file.

The ANSI Fortran Standard does not provide for the file identifier on BACKSPACE, ENDFILE, or REWIND statements.

7.11.2.1 BACKSPACE statement

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

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

7.11.2.2 ENDFILE statement

An ENDFILE statement writes an endfile record as the next record of the file; the file is then positioned after the endfile record, and the remainder of the file is truncated. A BACKSPACE or REWIND statement must be used before the file can be read. In environments where multi-file structures are supported (such as COS), the same unit can still be written to, in a new file that is positioned after the one just ended; see examples in 7.3.3.1 and 7.3.3.2.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

The ANSI Fortran Standard does not provide for the writing of an endfile on a file that is not connected.

7.11.2.3 REWIND statement

A REWIND statement causes the specified file to be positioned at its initial point. If the file is already positioned at its initial point, execution of the statement has no effect on the file position.

The ANSI Fortran Standard does not provide for the rewinding of an unconnected file or a file connected for direct access, nor does it provide for the creation of a connected file when one does not exist.

7.12 CHANGING MAXIMUM LENGTH FOR I/O LISTS AND FORMAT SPECIFICATIONS

Normally, all formatted I/O is restricted to an input/output record length of 267 characters on UNICOS or 152 characters on COS. Your site analyst can adjust these lengths within the I/O library used with CFT77. Or, if you use SEGLDR on a CRAY X-MP system, you can change the I/O buffer lengths at load time by performing the following steps:

- Specify the sizes of two common blocks that contain the working storage for formatted reads and writes.
- Change the values of two externally defined symbols that determine the maximum length of a formatted record.

SEGLDR directives COMMONS and SET can be used to make these changes. Common block $RFDCOM and symbol $RBUFLN determine the size of the buffer used for formatted reads. Common block $WFDCOM and symbol $WBUFLN determine the size of the buffer used for formatted writes. The size of either common block must be 9 words greater than the value of the symbol that defines the maximum size.

For example, to increase the maximum record length for formatted writes to 522 characters and the maximum record length for formatted reads to 384 characters, use the following SEGLDR directives:

```
SET=$WBUFLN:522
COMMONS=$WFDCOM:531
SET=$RBUFLN:384
COMMONS=$RFDCOM:393
```

See the Segment Loader (SEGLDR) Reference Manual, CRI publication SR-0066, for further information on the SET and COMMONS directives.

INPUT/OUTPUT FORMATTING 8

CFT77 allows both formatted and unformatted input/output (I/O). Unformatted I/O is considerably faster than formatted I/O.

CFT77 provides two methods of formatting program input and output. *List-directed I/O* is easy to use, but allows limited control over the format of input and output records. *Formatted I/O* permits detailed specification of data formats. Unformatted, formatted, and list-directed I/O are introduced with examples in the first part of section 7.

8.1 UNFORMATTED I/O

Unformatted I/O does not convert data from its internal (binary) representation and is not suitable for printing or for use by other vendors' computers. It is fast and retains the full precision of any numbers used by a program. Unformatted I/O uses blocked files under both UNICOS and COS, and can write unblocked COS datasets (see 7.3.1).

In a data transfer statement, a control information list showing no format specifier causes an unformatted transfer. Example:

```
WRITE(9)X
```

The above statement writes one record containing the value of X to the file connected to unit 9.

With unformatted I/O, all items in the input or output list are written to or read from a single record. Example:

```
WRITE(9)X,Y,Z
```

The above statement writes one record containing the values of X, Y, and Z to the file connected to unit 9.

List items are written and read based on their type. Integers, real numbers, and logical values each occupy one word of memory; complex and double-precision numbers each occupy two words of memory; and character strings are stored eight characters per memory word, left-justified. Unformatted READ statements read the appropriate number of memory words for the type of each variable in the input list.

Example:

```
LOGICAL TEST
CHARACTER*24 STRING
INTEGER Z
...
READ(9)TEST,STRING,Z
```

In the above program, one word from the file connected to unit 9 would be assigned to the variable TEST, the next three words would be assigned to STRING, and the next word would be assigned to Z.

Implied-DO lists can be used in unformatted I/O statements. Example:

```
WRITE(9)(VECTOR(I),I=1,55)
```

The above statement writes one record, containing the values of VECTOR(1) through VECTOR(55), to the file connected to unit 9.

8.2 LIST-DIRECTED I/O

List-directed I/O allows data formatting to be performed according to the type of the list item instead of by a format identifier. List-directed records consist of values and value separators. Each value is either a constant, a null value, or one of the following forms.

*r***c* *r**

r Unsigned, nonzero, integer constant

The *r***c* form is equivalent to *r* successive appearances of the constant *c*. The *r** form is equivalent to *r* successive null values. Neither of these forms can contain embedded blanks, except where permitted within the constant *c*.

Example of list-directed I/O, as shown in section 7 introduction:

```
WRITE (7,*) A,B,C,'   Best: ',BST          ! To unit 7
```

The above statement would give the following result:

```
3.316624790355, 2.236067977, 4.   Best: 4.
```

8.2.1 LIST-DIRECTED INPUT

The form of a list-directed input value must be acceptable for the type of the input list item. Blanks cannot be used as zeros. Embedded blanks are permitted only in complex constants and character constants.

In an input file to be read, value separators can have one of the following forms:

- A comma optionally preceded and followed by one or more contiguous blanks
- A slash optionally preceded and followed by one or more contiguous blanks
- One or more contiguous blanks between two constants or following the last constant

Notice that record separators (commonly created by the RETURN key on a terminal) do not serve as value separators for list-directed input.

Real or double-precision list items must be numeric and suitable for F formatting (see 8.5.4).

A complex list item consists of an ordered pair of numeric fields separated by a comma and enclosed in parentheses. The first numeric field is the real portion of the complex constant; the second numeric field is the imaginary portion. An end-of-record can occur between the real portion and the comma or between the comma and the imaginary portion. Each numeric field can be preceded or followed by blanks.

A logical list item must not include either slashes or commas among the optional characters permitted for L formatting.

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 to as many records as needed. A blank, comma, and slash can appear in character constants.

For example, for item length *len* and character constant length *w*, if $len \le w$, the leftmost *len* characters of the constant are read into the list item. If $len > w$, 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
- In character values

8.2.2 LIST-DIRECTED OUTPUT

The output format for list-directed I/O is the same as that required for input, except as noted below. The values are separated by one of the following.

- One or more blanks
- A comma optionally preceded or followed by one or more blanks

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.

New records begin as necessary but, except for complex and character constants, the end of a record does not occur within a constant and blanks do not appear within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I*w* descriptor, for some value of *w*.

Real and double-precision constants are produced with the effect of either an F descriptor or an E descriptor, depending on the magnitude *x* of the value and a range $10^{-2466} \le x < 10^{2466}$. If the magnitude *x* is within this range, the constant is produced with 0PF*w.d*; otherwise 1PE*w.d*E*e* is used (where *n*P specifies a change in the constant's magnitude). Reasonable values of *w*, *d*, and *e* are used for each of the cases involved.

A complex constant is enclosed in parentheses, with a comma separating the real and imaginary portions.

A character constant is not delimited by apostrophes and is not preceded or followed by a value separator. Each internal apostrophe in a character constant is represented externally by one apostrophe. For carriage control, the processor inserts a blank character in a character constant at the beginning of any record that begins with the continuation of a character constant from the preceding record.

List-directed formatting does not produce slashes as value separators or null values.

Each output record begins with a blank character for carriage control when the record is printed.

8.3 FORMATTED I/O

Formatted I/O writes and reads data in the form of ASCII characters, and specifies the format of the data. A *format specification* provides explicit formatting information to direct the formatting of data between its internal representation and corresponding character strings. A format specification can be given in a FORMAT statement, whose statement label (or a variable representing the label) is used as a format specifier in transfer statements. Example:

```
      WRITE(7,50)A,B
   50 FORMAT(/,E5.2,3X,E4.1)
```

In the above WRITE statement, 50 refers to the FORMAT statement that follows it.

Otherwise a format can be included in a transfer statement as a character expression. See 7.6, 7.7, 7.7.3.3, and pages 7-5 through 7-7.

A format is enclosed in parentheses. One format can contain another format; such nesting can be carried to nine levels. Character data following the right parenthesis of a format is ignored only when the format is contained in an array.

The ANSI Fortran Standard does not limit nesting of format specifications.

8.3.1 FORMAT STATEMENT

The FORMAT statement is a nonexecutable statement that allows more than one data transfer statement to use the same format specification. The statement label of a FORMAT statement is used as the format specifier in a READ, WRITE, or PRINT statement. A FORMAT statement must appear in the same program unit as any statement that references it.

label FORMAT (*flist*)

label Statement label (required)

flist List of items, each having one of the following forms:

ned
[*r*]*ed*
[*r*](*flist*)

ned Nonrepeatable descriptor

ed Repeatable descriptor

r Nonzero, unsigned integer constant called a repeat specification; if not specified, a value of 1 is assumed.

Commas can separate list items in *flist* but are required only under the following conditions.

- Between two adjacent digits which belong to different list items
- Between two adjacent apostrophe or quotation mark delimiters of separate format 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 or after the slash or the colon descriptor or between a P 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)
```

NOTE

To maintain compatibility with CFT and to prevent ambiguity, CFT77 interprets any labeled statement beginning with the character string **FORMAT(** as a FORMAT statement. That is, this cannot be part of an assignment statement to an element of an array named FORMAT, even though this would be allowed by the ANSI standard.

8.3.2 INTERACTION BETWEEN I/O LISTS AND FORMAT SPECIFICATIONS

The beginning of execution of a formatted I/O statement initiates format control. Each action of format control depends on information from the next descriptor provided by the format specification, and the next item in the I/O list, if one exists.

If a statement has an I/O list, at least one repeatable descriptor must exist in the format specification.

An empty format specification of the form ( ) can be used unless contained within another format specification. An empty format specification causes one input or internal record to be skipped or one output or internal record containing no characters to be written. No I/O list items can correspond to an empty format specification.

Except for repeated descriptors and embedded format specifications, a format specification is interpreted from left to right.

An embedded format specification or format descriptor preceded by an *r* is processed as a list of *r* format specifications or descriptors. An omitted repeat specification is treated the same as a repeat specification with a value of 1.

Each repeatable descriptor interpreted in a format specification corresponds to one item specified by the I/O list, except that an item of type complex requires the interpretation of two F, E, D, G, A, or R format descriptors. An I/O list contains no items corresponding to nonrepeatable format descriptors.

When format control encounters a repeatable format descriptor, it determines whether the I/O list has specified a corresponding item. If it has, format control transmits appropriately formatted information between the item and the record, then proceeds. If no corresponding item exists, format control terminates.

Format control also terminates if the rightmost parenthesis of a complete format specification is encountered and no additional I/O list items are specified. If another list item is specified, the file is positioned to the next record and format control reverts to the beginning of that format specification terminated by the next-to-last right parenthesis. If there is none, format control reverts to the first left parenthesis of the complete format specification. If reversion occurs, the reused portion of the format specification must contain at least one repeatable 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 8.5.6) or on S, SP, SS, BN, or BZ.

Examples:

In the following examples, the ↑ indicates the reversion point if list items remain when format control encounters the closing parenthesis.

```
1  FORMAT(10F10.3,1PE20.6)
         ↑

2  FORMAT(10F10.3,(1PE20.6))
                  ↑

3  FORMAT(I10,3(I5,2(I5,I7),3(L1,L2),I7))
              ↑

4  FORMAT(I5,2(I4,I6),3(I1,I2))
                      ↑
```

8.3.3 POSITIONING BY FORMAT CONTROL

If a T or X descriptor is the first 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 descriptor or an H, apostrophe, or quotation mark 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 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 descriptor is processed.

After a read operation, any unprocessed characters of the record read are skipped.

When format control terminates, the file is positioned after the current record.

8.4 FORMAT DESCRIPTORS SUMMARY

Format 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 repeatable, shown in table 8-1, or nonrepeatable, shown in table 8-2. A repeatable descriptor can be preceded by an integer; for example, 6F5.2 indicates 6 consecutive F fields that are 5 characters wide.

The legend for the following descriptors is on the following page.

Table 8-1. Repeatable Format Descriptors

Data Type	Format Descriptor	Description
Real	Ew.*d*	Real with exponent
	Ew.*d*E*e*	Real with specified exponent length
	Fw.*d*	Real without exponent
	Gw.*d*	Real with or without exponent
	Gw.*d*E*e*	Real with or without exponent or length
Double precision	Dw.*d*	Double-precision with exponent
Integer	Iw	Base-10 integer
	Iw.*m*	Base-10 integer, minimum number of digits
Character	A	Character with data-dependent length
	Aw	Character with specified length
	Rw	Right-justified character with length
Boolean	Ow	Octal integer
	Ow.*m*	Octal integer with leading zeros and minimum number of digits
	Zw	Hexadecimal integer
	Zw.*m*	Hexadecimal integer with leading zeros and minimum number of digits
Logical	Lw	Logical
New line	/ †	Causes the beginning of a new record.

† The / descriptor is nonrepeatable as described in the ANSI standard.

Legend for descriptors in table 8-1:

- *w* Field width in number of character positions in external record; including leading blanks, + or - , decimal point, and exponent.
- *d* Number of digits to the right of the decimal point within the field. On output all numbers are rounded.
- *e* Number of digits in the exponent; must not be greater than 6.
- *m* Minimum number of digits to be output.

All of the above values must be unsigned integer constants. *w* and *e* cannot be zero.

Table 8-2. Nonrepeatable Format Descriptors

Function	Format Descriptor	Description
Real-number magnitude	*k*P	Scale factor for later F, E, G, and D descriptors. Multiplier is 10^k.
Character output	" or '	Literal string within format
Hollerith Data	*n*H	Output Hollerith string
Blank Control	BN	Blanks ignored
	BZ	Blanks treated as zeros
Format Control	:	Terminate format control
Position Control	T*n*	Character position within record
	TR*n*	Position forward
	TL*n*	Position backward
	*n*X	Position forward
	$	Suppress carriage control
Plus sign control	SP	Include plus signs on positive values
	SS,S	Suppress plus signs

k Scale factor
n Number of positions: positive nonzero integer

Table 8-3 describes the usage of the CFT77 format descriptors with data types. * indicates legal usage for input and output. + indicates legal usage for output. - indicates illegal usage.

Table 8-3. Format Descriptors with Data Types

Data Types	Format Descriptors									
	I	F	E	D	G	L	A	O	Z	R
Character	-	-	-	-	-	-	*	-	-	-
Complex	-	*	*	*	*	-	*	*	*	*
Double-precision	-	*	*	*	*	-	-	+	-	-
Integer	*	-	-	-	-	-	*	*	*	*
Logical	-	-	-	-	-	*	*	*	*	-
Real	-	*	*	*	*	-	*	*	*	*

Format restrictions for integer, logical, and real variables can be lifted using SEGLDR and its EQUIV directive. To change the limitations for read and write operations, specify EQUIV=$RNOCHK($RCHK) or EQUIV=$WNOCHK($WCHK), respectively. Both of these EQUIV statements must be specified if changes are desired. Table 8-4 describes the format descriptors and data types when SEGLDR and the EQUIV directive is used. * indicates legal usage for input and output. - indicates illegal usage.

Table 8-4. Format Descriptors and Data Types when SEGLDR and the EQUIV Directive are Used

Data Types	Format Descriptors									
	I	F	E	D	G	L	A	O	Z	R
Integer	*	*	*	-	*	*	*	*	*	*
Logical	*	*	*	-	*	*	*	*	*	*
Real	*	*	*	*	*	*	*	*	*	*

8.5 FORMATTING REAL NUMBERS (F, E, G, D)

This subsection describes I/O formatting of real numbers, including double precision. Real values can be formatted with the F, E, and G descriptors; their use for output is described in 8.5.1 through 8.5.3. Because these descriptors have the same effect on input, their use for input is described separately in 8.5.4.

The following general rules apply to output of real values:

- A positive or zero internal value in the field is prefixed with blank characters except as described in 8.8.5 for plus-sign control (S descriptors). A negative internal value in the field is prefixed with blank characters followed by a minus sign.
- The value is right-justified in the field. If the number of characters produced by the formatting is smaller than the field width, leading blanks are inserted in the field.
- If the number of characters exceeds the field width, the entire field is filled with asterisks.

Real output using the F, E, and G descriptors lets you choose between scientific and conventional notation; G results in the use of scientific notation only when the output field cannot otherwise accommodate a given value. Table 8-5 illustrates how these descriptors affect output.

Table 8-5. Real Output Values with F, E, and G Descriptors

Internal	(F12.5)	(E12.5)	(G12.5)
269181.0	269181.00000	0.26918E+06	0.26918E+06
-269181.0	************	-0.26918E+06	-0.26918E+06
26918.1	26918.10000	0.26918E+05	26918.
-41.27	-41.27000	-0.41270E+02	-41.27
0.38176	0.38176	0.38176E+00	0.38176
0.038176	0.03817	0.38176E-01	0.38176E-01
0.00038176	0.00038	0.38176E-03	0.38176E-03

8.5.1 REAL OUTPUT WITHOUT EXPONENT (F)

Use the F*w*.*d* descriptor to express a real value as a decimal number with no exponent. The value occupies *w* positions, and the fractional portion consists of *d* digits.

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. The value is altered by an applicable 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:

Internal Representation	F Descriptor	Output Field Positions 1 2 3 4 5 6 7 8 9 10
3.1415926	F10.5	3 . 1 4 1 5 9 (positions 4–10)
-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 (positions 2–8)
0	F7.6	. 0 0 0 0 0 0

The ANSI Fortran Standard does not specify output formatting for values too large to be printed in the specified field.

8.5.2 REAL OUTPUT WITH EXPONENT (E)

Use the Ew.*d* and Ew.*d*E*e* descriptors to represent real numbers in scientific notation; that is, with exponents. *w* indicates that the external field occupies *w* positions. The exponent portion consists of *e* digits. 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 output field in the absence of the P descriptor (see 8.5.5) is as follows:

[-][0].x_1 x_2 ... x_{d-1} x_d *exp*

$x_1 x_2$ ... x_d
The most significant digits of the rounded data

exp Decimal exponent of one of the following forms

Table 8-6 shows the forms used for exponents with E formatting. Each *n* is a single digit. If $|exp| \geq 1000$, the entire field is shifted left one position to provide for n_4; if space has not been provided, the entire field is replaced with asterisks. If *e* is greater than the number of digits necessary to express *exp*, leading zeros are inserted. $w \geq d+5$.

Table 8-6. Output of Exponents with E Descriptor

Descriptor	Absolute Value of Exponent	Output Form of Exponent
Ew.*d*	*exp* = 0	E+00
Ew.*d*	$0 < \|exp\| \leq 99$	E$\pm m_1 m_2$
Ew.*d*	$100 \leq \|exp\| \leq 999$	$\pm m_1 m_2 m_3$
Ew.*d*	$1000 \leq \|exp\| \leq 2466$	$\pm m_1 m_2 m_3 m_4$
Ew.*d*E*e*	$\|exp\| \leq (10**e)-1$	E$\pm m_1 m_2 m_3 ... n_e$

Internal Representation	E Descriptor	Output Field Positions 1 2 3 4 5 6 7 8 9 10 12
365.26	E10.2	0 . 3 7 E + 0 3
-365.26	E11.5	- . 3 6 5 2 6 E + 0 3
.000000099	E11.3	0 . 9 9 0 E - 0 7
100.	E11.2E1	. 1 0 E + 3
100.	E11.2E4	. 1 0 E + 0 0 0 3

8.5.3 REAL OUTPUT WITH OPTIONAL EXPONENT (G)

Use the G*w*.*d* and G*w*.*d*E*e* descriptors to output real numbers in scientific notation only when the field cannot accommodate conventional notation. *w*, *d*, and *e* have the same meanings as in F formatting. The total output field width *w* must be at least 6 spaces wider than the fractional value *d*, to accommodate the decimal point, signs, and exponent notation.

Each value's representation depends on its magnitude. Table 8-7 shows how values are formatted for a magnitude of M. The number of spaces (*n*X) is 4 for descriptors with no E*e* value, and *e*+2 when E*e* is present. *k* is the scale factor, which is effective only if a value's magnitude exceeds the range for F formatting.

Table 8-7. G*w*.*d* and G*w*.*d*E*e* and Equivalent F or E Descriptors

Magnitude of Data	Equivalent Descriptors
$0.1 \leq M < 1$	F(*w*-*n*).*d*,*n*X
$1 \leq M < 10$	F(*w*-*n*).(*d*-1),*n*X
. . .	. . .
$10^{d-2} \leq M < 10^{d-1}$	F(*w*-*n*).1,*n*X
$10^{d-1} \leq M < 10^{d}$	F(*w*-*n*).0,*n*X
$M < 0.1$ or $M \geq 10^{d}$	*k*P,E*w*.*d*

Examples:

```
   Internal           G          Output Field Positions
Representation    Descriptor    1 2 3 4 5 6 7 8 9 10 11 12

-324.876            G12.6       - 3 2 4 . 8 7 6
.487295343397       G10.5       . 4 8 7 3 0
-72.59              G10.3         - 7 2 . 6
.000000000019       G12.2                 . 1 9 E -  1  0
.000000000019       G9.1              . 2 E - 1 0
10000.              G12.2                 . 1 0 E +  0  5
10000.01            G12.2                 . 1 0 E +  0  5
10000.              G12.2E1                 . 1 0 E  +  5
10000.              G12.2E4           . 1 0 E + 0 0  0  5
```

8.5.4 REAL INPUT (F, E, G)

This subsection describes the input of real values with the F, E, and G descriptors, which behave identically for input. The following rules apply to real input formatting:

- Leading blanks are not significant.
- Plus signs can be omitted.
- A field of all blanks has the value 0.
- A decimal point appearing in the input field overrides the portion of a descriptor that specifies the decimal point location.
- The input field can have more digits than are used by the processor in approximating the value of the data. The excess digits are used for round-off but are otherwise discarded.

The E, F, and G descriptors take the following forms:

```
Fw.d

Ew.d    Ew.dEe

Gw.d    Gw.dEe
```

w Total positions in the input field

d Number of digits in the fractional portion. If the input value contains more than this number of digits, the number of digits that fit in the total field length are stored. (The number of input fractional digits includes not only those shown explicitly, but also those resulting from an exponent in the input value or from an applicable *k*P scale factor.)

e Number of digits in the exponent

The input field consists of an optional sign followed by a string of digits including an optional decimal point, followed by an optional exponent. The exponent can take 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.

Examples:

Input Field Positions 1 2 3 4 5 6 7 8 9 10	Format Descriptor	Internal Representation
1 7 8 7 . 1 9 8 7	F9.4	1787.1987
- 1 7 8 7 . 1 9 8 7	F10.4	-1787.1987
6 . 0 2 3 D 2 3	F8.3	602300000000000000000000.
+ 1 0 4 8 5 7 5 . 7 5	E11.2	1048575.75
1 . 5 9 2 E 3	E12.3	1592.

Input Field Positions 1 2 3 4 5 6 7 8 9 10	Format Descriptor	Internal Representation
- 3 2 . 7 6 8 D 0 4	E10.3	-327680.
+ 8 7 8 . 4 9 2 1	G9.4	878.4921

When they fit in the total field width, extra fractional digits in the input take precedence over the *d* fractional value in the descriptor:

Input Field Positions	Format Descriptor	Internal Representation
- 1 7 8 7 . 1 9 8 7	F10.3	-1787.1987
- 1 0 4 8 5 7 5 . 7 5	E11.0	-1048575.75
4 7 2 1 . 0 E - 2	G12.1	47.21
- . 6 2 9 0 0 0 0	G10.2	-.629

Rightmost decimal digits are truncated when the total field length is too short to accommodate them:

Input Field Positions	Format Descriptor	Internal Representation
- 1 7 8 7 . 1 9 8 7	F9.4	-1787.198

If you omit the decimal point, the compiler uses the rightmost *d* digits of the string as the fractional part of the value, with leading zeros assumed if necessary. This requires that format specifications be perfectly matched to the form of your input data and therefore increases the risk of invalid data. To make sure your data is correct, it is suggested that you include explicit decimal points.

Examples:

Input Field Positions 1 2 3 4 5 6 7 8 9 10 12	Format Descriptor	Internal Representation
1 9 8 7	F4.0	1987.
1 9 8 7	F4.4	.1987
1 9 8 7	F2.0	19.
6 5 5 3 6 E - 5	E8.3	.00065536
3 8 (positions 10–11)	E11.11	.00000000038
- 1 4 9 2 E - 3	F8.0	-1.492
7 2 D 1 0	G5.0	720000000000.

8.5.5 DOUBLE-PRECISION (D)

D formatting is used for double-precision real numbers; its use is identical to E formatting.

8.5.5 SCALE FACTOR (P)

The P descriptor takes the form *k*P, where *k* is an optionally signed integer constant called the *scale factor*. *k*P represents 10^k as a multiplier.

The scale factor is 0 at the beginning of each I/O statement. It applies to all subsequently interpreted F, E, D, and G 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 formatting in the following ways:

- With F, E, D, and G input formatting (provided that no exponent exists in the field) and with F output formatting, 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 formatting, the scale factor has no effect if there is an exponent in the field.
- On output with E and D formatting, 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 formatting, the effect of the scale factor is suspended unless the magnitude of the data to be formatted requires the use of E formatting. In this case, the scale factor has the same effect as with E output formatting.

Examples:

Input field:		9876.54	98.7654E2	9876.54	987.654
FORMAT statement:	FORMAT (	2PF8.3,	-2PE9.4,	F9.4,	0PG9.4)
Internal representation:		98.7654	9876.54	987654.	987.654

Internal representation:		9.87654	9876.54	9876.54	987.654
FORMAT statement:	FORMAT (	2PF12.2,	-2PE12.4,	F12.4,	1PG12.2)
Output field:		987.65	0.0099E+06	98.7654	9.88E+02

Scale factor k controls decimal normalization. If $-d \le k \le 0$, there are $|k|$ leading zeros and $d-|k|$ significant digits after the decimal point. If $0<k<(d+2)$, there are k significant digits to the left of the decimal point and $d-k+1$ significant digits to the right of the decimal point. Other values of k are not permitted.

8.6 FORMATTING OTHER DATA TYPES

This subsection presents descriptors that are used for integer, complex, and logical values, and bit strings in octal or hexadecimal form.

8.6.1 INTEGER (I)

The Iw and Iw.*m* descriptors are used for integers, and indicate that the field to be formatted occupies w positions. The specified I/O list item must be of type integer. On input, the specified list item becomes defined with an integer value. On output, the specified list item must be defined with an integer value.

In the input field, the character string must be in the form of an optionally signed integer constant. Leading blanks in the input field are ignored. The Iw.*m* descriptor is treated identically to the Iw descriptor.

The output field for the Iw descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. If the value (plus the possible minus sign) exceeds w digits, the field is filled with asterisks.

If the Iw.*m* 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 (‿ represents a blank character):

```
      READ 20,I,J,K
  20  FORMAT(I2,I5,I3)

      15‿‿-10‿‿                    ← input line

      PRINT 10,I,J,K
10    FORMAT(I5,I3,I4)
```

Execution of the above results in the following printout:

```
      ‿‿‿15-10‿‿‿0
```

8.6.2 COMPLEX

Complex data consists of a pair of separate real data. Data formatting must be specified by two successively interpreted A, D, E, F, G, O, R, or Z descriptors. The first of the descriptors specifies formatting for the real part; the second for the imaginary part. The two descriptors can differ. Nonrepeatable descriptors can appear between two successive A, D, E, F, G, O, R, or Z descriptors.

8.6.3 LOGICAL (L)

The L*w* descriptor indicates processing of a logical list item and an input or output field width of *w* positions. The specified I/O list item must be of type logical. On input, the list item becomes defined with logical data. On output, the list item must be defined with logical data.

The input field consists of a T for true or an F for false, optionally followed by additional characters. The field can contain a leading period or leading blanks.

The output field consists of *w*-1 blanks followed by a T or F, depending on the value of the internal data.

Examples:

Input Field Positions												L	Internal
1	2	3	4	5	6	7	8	9	10		12	Descriptor	Representation
T												L1	(true)
.	T	R	U	E								L4	(true)
		F										L3	(false)
		.	F	A	L	S	E					L12	(false)
				T	1	2	3					L7	(true)
					F	A	B	C				L9	(false)
									.	T		L12	(true)
										.	F	L12	(false)

Internal	L	Output Field Positions											
Representation	Descriptor	1	2	3	4	5	6	7	8	9	10		12
(true)	L6						T						
(false)	L12												F
(true)	L10										T		
(false)	L1	F											
(true)	L1	T											
(false)	L3			F									

8.6.4 OCTAL (O) (CFT77 EXTENSION)

The Ow descriptor specifies octal representation of a 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 Ow descriptor for output only. The octal value represents the bit content of the list item in memory, not its numeric value as used within a program.

Example:

```
R=10.0
I=10
WRITE(*,'(O22/O22)')R,I
```

The above code writes values R and I as follows:

```
0400045000000000000000
0000000000000000000012
```

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 represents a bit pattern that is interpreted depending on the data type of the value, as shown in appendix G.

The value is right-justified in the list item if fewer than 22 octal digits are contained in the field. Unspecified bit positions are cleared to 0. A blank field is considered to be a field containing all zeros. If the first nonblank character in the field is a minus, the ones complement of the value is stored.

On output, the internal representation of the list item is converted to octal and the rightmost w octal digits are right-justified in the field.

If the list item is not of type double-precision and the field is larger than 22 positions, the output contains leading blank characters. If the list item is of type double-precision and w is greater than 45, the output contains leading blank characters. If w is greater than 22, a blank character occupies position (w-22) in the output field. This character indicates the beginning of the double-precision portion. To completely output a double-precision value, the value of w must be at least 45.

8.6.5 HEXADECIMAL (Z) (CFT77 EXTENSION)

The Zw descriptor specifies hexadecimal representation of a list item of type integer, real, complex, Boolean, or logical and a field width of *w* positions. The hexadecimal value represents the bit content of the list item in memory, not its numeric value as used within a program.

Example:

```
R=10.0
I=10
WRITE(*,'(Z16/Z16)')R,I
```

The above code writes values R and I as follows:

```
4004A00000000000
000000000000000A
```

On input, the field contains a string of from 0 to 16 hexadecimal characters representing a value to be stored into the list item. This value represents a bit pattern that is interpreted depending on the data type of the value, as shown in appendix G.

This value is right-justified in the list item if fewer than 16 hexadecimal characters are contained in the field; leading zeros are assumed. A blank field is assumed to be a field of all zeros. If the first nonblank character in the field is a minus, the ones complement of the value is stored.

On output, the internal representation of the list item is converted to a zero or positive hexadecimal value and the rightmost *w* digits are right-justified in the field. If the field is larger than 16 positions, leading blank characters are output.

8.7 FORMATTING CHARACTER DATA (A, ', ", H)

Most transfers of character data use the A descriptor. Single and double quotation marks allow a literal string to be included directly in a format line. H is used for output of Hollerith data.

8.7.1 CHARACTER TYPE (A)

Use the A[w] descriptor for I/O list items of type character. (See appendix E.3.3 concerning the use of Aw with other data types.) There are three character lengths to consider:

- *len* is the character length of the list item, as declared in your program by CHARACTER**len*.
- *w* is the character length of the transfer field, as in the format descriptor A8; if *w* is not specified, the field has the same length as the list item. The relation between *w* and *len* is described in the next paragraph.
- For input, the length, *s*, of a string to be read might differ from *w*. If $w < s$, only the leftmost *w* characters are read. If $w > s$, the string is left-justified in the field.

If $len \neq w$, the data is handled as follows:

	$w > len$	$w < len$
On input:	The rightmost *len* characters of the input field are stored.	Input characters are left-justified in storage, followed by blanks.
On output:	The item from storage is right-justified within the output field, preceded by blanks.	Field contains the leftmost *w* characters from storage.

Input examples for a list item declared as CHARACTER*6:

Input Positions 1 2 3 4 5 6 7 8	A Descriptor	Storage Positions 1 2 3 4 5 6 7 8	Case
A B C D E F G H	A or A6	A B C D E F	$w = len$
A B C D E F G H	A4	A B C D	$w < len$
A B C D E F G H	A8	C D E F G H	$w > len$

Output examples for a list item declared as CHARACTER*6:

Storage positions 1 2 3 4 5 6 7 8	A Descriptor	Output positions 1 2 3 4 5 6 7 8	Case
A B C D E F	A or A6	A B C D E F	w = *len*
A B C D E F	A4	A B C D	w < *len*
A B C D E F	A8	A B C D E F (in positions 3–8)	w > *len*

8.7.2 OUTPUT STRINGS WITHIN FORMAT LINES (' , ")

Use the apostrophe or quotation mark descriptor to include a literal character string, including blanks, in a format specification. These 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 include the quotation mark descriptor.

Example:

```
      INTEGER J,B,G
      ...
      WRITE(*,12) J,B,G
   12 FORMAT(' John''s:  ',I2/," Bill's:  ",I2/,' "Great": ',I2)
```

Execution of the above produces the following printout:

```
 John's:  4
 Bill's:  6
 "Great": 10
```

8.7.3 HOLLERITH CHARACTER OUTPUT (H)

The *n*H descriptor causes character information to be written from the *n* characters (including blanks) following the H of the descriptor. An H descriptor can be used only for output. Hollerith is discussed in E.1.

Example:

```
      WRITE(41,16)
   16 FORMAT(' LABEL',5H UNIT,' 41')
```

8.8 SPECIAL-PURPOSE DESCRIPTORS (T, X, /, :, B, S, $)

This subsection describes special-purpose descriptors for position control, the use of blanks, the use of plus signs, printer control, and format control.

8.8.1 POSITION CONTROL (T, TL, TR, X)

The T, TL, TR, and X descriptors specify the position where the next character will be transmitted to or from within the record. This allows portions of a record to be written or read more than once, possibly with different formatting.

The T*c* descriptor specifies an absolute character position, *c*, within the record. The TL, TR, and X descriptors specify a character position relative to the current position: TL*n* moves *n* spaces to the left, and TL*n* or *n*X moves *n* spaces to the right. If *n* in TL*n* exceeds the current character position, the new position is the first character. If the use of T, TR, or X specifies a position past the end of the record, the record is extended to include that position, and the extra positions are filled with blanks.

These descriptors allow you to replace a character that is already in the record, without affecting other characters. The new position is the position following the most recently written position.

Example:

```
CHARACTER*1 N,S,JR*2,NAME*12
NAME='norm swenson'
N='N'
S='S'
JR='Jr'
WRITE(*,'( 1X, A12,   T2, A1,  T7, A1, T15, A2 )') NAME,N,S,JR
WRITE(*,'( 1X, A12, TL12, A1,  4X, A1,  7X, A2 )') NAME,N,S,JR
WRITE(*,'( 1X, A12, TL12, A1, TR4, A1, TR7, A2 )') NAME,N,S,JR
```

All three WRITE statements above write the following record:

```
 Norm Swenson Jr
```

The formats shown include an initial blank space for printer control. This space must be accounted for in the use of the T descriptor. The space between Swenson and Jr is not included in character values NAME or JR; it results only from the positioning specified by the descriptors. Notice that the TR and X descriptors use the same character counts and have the same effects.

8.8.2 END OF RECORD (/)

The slash 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 formatting of *n* adjacent records can be specified by the appearance of *n* consecutive slashes (optionally separated by commas) or by preceding a single slash with a value, *n*, equal to the number of records to be processed.

The ANSI Fortran Standard does not provide for a repeat count for slash formatting.

Examples:

```
      PRINT 39
   39 FORMAT('1LINE 1',/,' LINE 2'/' LINE 3'///' LINE 6')


      READ(99,42) RECORD3
   42 FORMAT(2/,...)
```

8.8.3 TERMINATE FORMAT (:)

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 descriptor terminates the formatted transfer of data if no I/O list items remain to be processed. If unprocessed I/O list items remain, the colon descriptor has no effect on format control. Termination of format control by a colon descriptor causes the record being processed to become the preceding record.

Examples:

(1)

```
      PRINT 10,X
   10 FORMAT(' X= 'F10.5,' Y= 'F10.5)
```

Execution of the above results in the following printout:

```
      X= 1234.56789 Y=.
```

(2)

```
      PRINT 20,X
   20 FORMAT(' X= 'F10.5,:' Y= 'F10.5)
```

Execution of the above results in the following printout:

```
 X= 1234.56789.
```

8.8.4 INTERPRETING BLANKS (BN, BZ)

The BN and BZ descriptors specify whether blanks other than leading blanks are interpreted as nulls (ignored) or zeros. BN and BZ affect only input fields using the I, F, E, D, or G descriptors.

Either descriptor overrides the BLANK= specification in the file's OPEN statement, which defaults to NULL or BN. Notice that this default is the same for both opened files and default files, but programs ported from other vendors' systems may have been written on the assumption that default files use BZ as the default for blank interpretation. The ANSI standard does not specify a default.

The BN 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 descriptor causes nonleading blank characters to be treated as zeros.

Either descriptor applies to all subsequent descriptors in the specification until the next BN or BZ descriptor, if any. Example:

```
   40 FORMAT (BZ,F6.3,2X, BN,F5.1,2X,F9.7)
   50 FORMAT (F6.3,2X, BZ,F5.1,2X,F9.7)
```

In the first statement above, BZ applies to the first F descriptor; the BN descriptor applies to the second and third F descriptors. In the second statement, the default value of BN applies until the BZ descriptor, which applies to the second and third F descriptors.

8.8.5 PLUS SIGN CONTROL (S, SP, SS)

The S, SP, and SS descriptors control plus signs in numeric output fields. Normally, the compiler suppresses plus signs. The SP descriptor causes plus signs to be produced on numeric output fields until either an S or an SS descriptor is encountered. The SS descriptor specifies suppression of plus signs; the S descriptor restores the normal compiler option, which, in this case, is also the suppression of plus signs.

8.8.6 CARRIAGE CONTROL ($) (COS ONLY, CFT77 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:

```
      WRITE (6,100)
 100  FORMAT(' WHAT IS YOUR NAME?',$)
      READ (5,105) NAME
 105  FORMAT (4A8)
```

Execution of the above results in the following printout:

```
      WHAT IS YOUR NAME?
```

The response (in this example, HARRY) can go on the same line:

```
      WHAT IS YOUR NAME? HARRY
```

CRAY I/O EXTENSIONS 9

This section describes nonstandard I/O features of CFT77: buffer in and buffer out, NAMELIST, and random I/O.

9.1 BUFFER IN AND BUFFER OUT STATEMENTS (CFT77 EXTENSIONS)

BUFFER IN and BUFFER OUT cause data to be transferred while allowing the subsequent execution sequence to proceed concurrently. BUFFER IN and BUFFER OUT operations can proceed simultaneously on several units or files. BUFFER IN is for reading; BUFFER OUT is for writing. A buffered operation includes only data from a single array or a single common block.

Either statement initiates a data transfer between the specified file or unit (at the current record) and the specified memory range. If the unit or file is completing a buffered I/O operation initiated earlier, BUFFER IN or BUFFER OUT suspends the execution sequence until the earlier operation is complete. At termination, execution of the BUFFER IN or BUFFER OUT statement completes as if no delay had occurred.

Under UNICOS, BUFFER IN and BUFFER OUT work only on pure data files. Under COS, they work on unblocked files (equivalent to pure data), and also on blocked; but blocked files are considerably slower, and BUFFER OUT cannot be used on blocked random datasets. The pure data/unblocked structure must be specified to the operating system, as shown in subsection 9.2.1. File structures are discussed in 7.3.1.

Buffered data transfers must be performed in multiples of 512 words, except those on COS blocked datasets.

Use the UNIT or LENGTH function to delay the execution sequence until the buffered I/O operation is complete (see 9.1.1 and 9.1.2). These functions can also return information about the I/O operation at its termination. You can use SETPOS with BUFFER IN and BUFFER OUT for random positioning; see 9.2.1 and the Programmer's Library Reference Manual, CRI publication SR-0113.

```
BUFFER IN (id,m) (bloc,eloc)

BUFFER OUT (id,m) (bloc,eloc)
```

id Identifier, one of the following:

- Unit identifier expressed as an integer or as a Hollerith expression of up to seven characters
- File name expressed as a character string or a character or integer variable containing Hollerith data of up to seven characters. The file identifier cannot be used on the CRAY-2 system.

m Mode identifier controls the record position following the data transfer; must be an integer expression. Discussed in the following text. The mode identifier is not used on the CRAY-2 system; only full-record processing is available.

bloc
eloc Symbolic names of the variables, arrays, or array elements that mark the beginning and end locations of the buffered I/O transfer. Must be either elements of a single array (or equivalenced to an array) or members of the same common block. Neither *eloc* nor *bloc* can be a character entity.

The mode identifier *m* controls the position of the record at unit *u* after the data transfer is complete. Values of *m* have the following effects:

- $m \geq 0$ causes full record processing, so that file and record positioning works as with conventional I/O. The record position following such a transfer is always between the current record (the record with which the transfer occurred) and the next record. BUFFER OUT with $m \geq 0$ ends a series of partial-record buffered output transfers.

- $m < 0$ causes partial record processing. In BUFFER IN, the record is positioned to transfer its (n+1)th word if the nth word was the last transferred. In BUFFER OUT, the record is left positioned to receive additional words.

The amount of data to be transferred is specified in Cray words without regard to types or formats. However, the data type of *eloc* affects the exact ending location of a transfer. If *eloc* is double-precision or complex, the location of the second word in its two-word form of representation marks the ending location of the data transfer.

BUFFER OUT with *bloc=eloc*+1 causes a zero-word transfer and concludes the record being created. Except for terminating a partial record, *bloc* following *eloc* in a storage sequence causes a run-time error.

Example:

```
      PROGRAM XFR
      DIMENSION A(1000), B(2,10,100), C(500)
      BUFFER IN(32,0) (A(1),A(1000))
      ...
      DO 10 J=1,100
        B(1,1,J) = B(1,1,J) + B(2,1,J)
   10 CONTINUE
      BUFFER IN(32,0) (C(1),C(500))
      BUFFER OUT(22,0) (A(1),A(1000))
      ...
      END
```

The BUFFER IN statement above initiates a transfer of 1000 words from unit 32. Processing unrelated to that transfer proceeds; when this is complete, a second BUFFER IN is encountered, 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 as the execution sequence continues. A BUFFER OUT begins the transfer of the first 1000 words to unit 22. *m*=0 in all cases, indicating full record processing.

9.1.1 THE UNIT FUNCTION

After execution of BUFFER IN or BUFFER OUT, the normal execution sequence continues concurrently with the data transfer. If the UNIT function is called in this execution sequence, the sequence is delayed until the transfer is complete. After the BUFFER IN operation, use the UNIT or LENGTH function before using memory locations where the data is stored.

When the transfer is complete, the UNIT function provides one of the following real data type values to the expression that calls it.

CRAY X-MP computer system:

Value	Meaning
-2.0	Partial record read operation (BUFFER IN with *m*<0) completed successfully without encountering the end of the current record
-1.0	Operation other than a partial read completed successfully
0.0	End-of-file was read
1.0	Disk parity error occurred during reading
2.0	Other device malfunction occurred during reading or writing

CRAY-2 computer system:

-1.0 Successful completion.

0.0 An end-of-file was read.

1.0 An error occurred.

Example:

```
      PROGRAM TESTUNIT
      DIMENSION M(200,5)
   10 BUFFER IN (32,0) (M(1,1),M(200,5))
      IF (UNIT(32) .GE. 0.0) GOTO 14
   11 DO13 J=1,5
        DO12 I=1,200
          M(I,J)=M(I,J)*2
   12   CONTINUE
   13 CONTINUE
      BUFFER OUT (22,0) (M(1,1),M(200,5))
      IF (UNIT(22) .LT. 0.0) GOTO 10
   14 END
```

9.1.2 THE LENGTH FUNCTION

If the LENGTH function is called during a buffered I/O operation, the execution sequence is delayed until the transfer is complete. LENGTH then returns the number of Cray 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 .GE. 0.0) GOTO 14
   11 DO 12  I=1,K,1
   12 IF (V(I) .EQ. 'KEY') GOTO 13
      IF (X .EQ. -2.0) GOTO 10
      STOP
   13 ...
      ...
   14 END
```

9.2 RANDOM INPUT/OUTPUT OPERATIONS (CFT77 EXTENSION)

Random access I/O offers high speed along with nonsequential access to any part of a file. As with direct access, (see 7.11.1), this allows you to modify the contents of a file without writing over the remainder of the file. Different kinds of random access I/O are performed by a set of Cray subroutines, which are documented in the Programmer's Library Reference Manual, CRI publication SR-0113. This is a summary of the available routines and their uses.

Table 9-1 shows comparative performance for different aspects of I/O operations, and table 9-2 shows the general characteristics of these routines. *Buffering* is a system activity that frees you from controlling the lengths of segments being transferred. An unbuffered transfer is faster because it frees the system from this function. This buffering is distinct from the concurrent execution allowed by the BUFFER IN and BUFFER OUT statements.

GETPOS and SETPOS are analogous to using a pointer that is set with the LOC function. This allows you to control the file position with variable record lengths, without the need to specify actual record numbers. GETPOS returns the current record number for a given file; SETPOS sets the record number for a file. GETPOS and SETPOS can be used with READ and WRITE or BUFFER IN and BUFFER OUT (see 9.2.1).

GETWA and PUTWA perform read and write operations that are word addressable, random access, and buffered. They can be synchronous or asynchronous. They are used only with the pure data/blocked file structure (see 7.3.1). The buffer defaults to 16 sectors or can be set up with the WOPEN routine. SEEK can be used to read ahead before GETWA.

READMS and WRITMS perform read and write operations that are record addressable, of variable record length, and buffered, using the default buffer size of 16 sectors (also changeable with WOPEN). They can be used asynchronously if ASYNCMS is called. In this case the calls to READMS and WRITMS must not be overwritten before the operation is completed; repeated calls to the same unit do not suspend processing. READMS and WRITMS are used only with the pure data/blocked file structure (see 7.3.1).

READDR and WRITDR perform read and write operations that are record addressable and unbuffered. These routines work on multiples of 512 words. SYNCDR makes them synchronous, and ASYNCDR makes them asynchronous; WAITDR waits until an I/O instruction has completed. For asynchronous use, the calls to READDR and WRITDR must not be overwritten before the operation is completed; repeated calls to the same unit do not suspend processing. READMS and WRITMS are used only with the pure data/blocked file structure (see 7.3.1).

Table 9-1. Performance Comparison of I/O Methods

	Blckd READ/ WRITE	Unblckd READ/ WRITE	Blockd BUFIN/ BUFOUT	Unblckd BUFIN/ BUFOUT	Direct Access	READMS WRITMS	READDR WRITDR	GETWA PUTWA
Striping[1]	C	E	B	A	D	D	A	D
Positioning	C	A	C	A	C	C/A[2]	A	C/A[2]
Random rewrite	D	A	E	A	D	C/A[2]	A	C/A[2]
File status chk	B	B	B	B	B	B	A	B
File statistics	C	C	C	C	C	A	A	A
Sequential spd.	A	C	A	C	B/D[3]	B	C	B
Data reusabilty	E	E	E	E	E	A	E	A
CPU overhead	B	A	B	A	B	D	A	D
Memory use	B	A	B	A	B	C	A	C

A through E indicate relative speeds for different operations. A is the fastest; E is the slowest.

1 Striping allows two operations to execute simultaneously. When striping is logical and user-driven rather than a hardware operation, the transfers must be asynchronous. (See the following example, in which two BUFFER IN or OUT operations occur at the same time.) BUFFER IN and BUFFER OUT are asynchronous; READDR and WRITDR also can be.

2 If record length is a multiple of 512, positioning and random rewrite are enhanced.

3 B for reading, D for writing

Table 9-2. Characteristics of Random I/O Methods

	READ, WRITE	BUFFER IN, BUFFER OUT	GETWA PUTWA	READMS, WRITMS	READDR, WRITDR
Sequential or Random	either	either[1]	either	either	either
Synchronous or Async.	Sync	Asynchron.	either	either	either[2]
Formatted or Unformtd	either	Unformatted	Unfmtd	Unfmtd	Unfmtd
Buffered blocked[3]	yes	yes			
Buffered unblocked[3]			yes	yes	
Unbuffered and unblckd	yes[4]	yes			yes

1 BUFFER IN and BUFFER OUT do not work on blocked random files.
2 Record length = 512 * n
3 This buffering is distinct from BUFFER IN/OUT. See preceding text.
4 Only unformatted

In the above table, the entry *yes* indicates that this combination of characteristics is available. A blank box indicates the combination is not available.

9.2.1 BUFFER IN/OUT WITH SETPOS

The following example shows two subroutines for reading and writing, offering flexibility and high performance. They use SETPOS with the BUFFER IN and BUFFER OUT statements, and have the following advantages:

- Large records do not involve extensive overhead and do not need buffer space.
- I/O is performed directly with the user array.
- Blocking and deblocking do not entail any CPU overhead.
- You can rewrite freely in the file space.
- Two I/O streams can be active simultaneously.
- The file is extendable.

This example works only with the pure-data/unblocked file structure. File structures are discussed in 7.3.1. For files FILEA and FILEB, this structure is specified as follows.

Under COS:

The following JCL statements should appear before the statement that runs your program (such as SEGLDR,GO.):

```
ASSIGN,DN=FILEA,U.
ASSIGN,DN=FILEB,U.
```

Under UNICOS:

(1) The following works with either shell.

```
...
env FILENV=asnfile assign -s u FILEA
env FILENV=asnfile assign -s u FILEB
...
env FILENV=asnfile a.out
```

(2) The following works with the Bourne shell:

```
...
FILENV=asnfile
export FILENV
assign -s u FILEA
assign -s u FILEB
...
a.out
```

(3) The following works with the C shell:

```
...
setenv FILENV=asnfile
export FILENV
assign -s u FILEA
assign -s u FILEB
...
a.out
```

In the example, I and M are record numbers; ISIZE is the record size. Each record holds one complete array. Subroutines FILLA, FILLB, and COMPUTE are not shown; the first two put initial data in the arrays, and the third changes this data.

```
      PARAMETER (NRECS = 100)              ! Number of records
      PARAMETER (ISIZE = 65536)            ! Must be multiple of 512
      REAL ARRAYA(ISIZE), ARRAYB(ISIZE)
      OPEN (UNIT=1,FILE='FILEA',ACCESS='DIRECT',FORM='UNFORMATTED')
      OPEN (UNIT=2,FILE='FILEB',ACCESS='DIRECT',FORM='UNFORMATTED')

C  Write initial data to two files simultaneously

      DO 10 I=1,NRECS
        CALL FILLA (ARRAYA,ISIZE,I)        ! Fill ARRAY with data to write
        CALL FILLB (ARRAYB,ISIZE,I)
        CALL RDMWRITE (1,ARRAYA,I,ISIZE)   ! ARRAYA to record I on unit 1
        CALL RDMWRITE (2,ARRAYB,I,ISIZE)   ! ARRAYB to record I on unit 2
   10 CONTINUE

C  Read random record N from both files and rewrite.  1≤N≤NRECS

      N = NRECS
      CALL RDMREAD (1,ARRAYA,N,ISIZE)
      CALL RDMREAD (2,ARRAYB,N,ISIZE)
      CALL IWAIT(1)                  ! Calls to IWAIT follow RDMREAD calls
      CALL IWAIT(2)                  !  so both streams can go in parallel
      CALL COMPUTE (ARRAYA,ARRAYB,N,ISIZE) ! Alter array contents
      CALL RDMWRITE (1,ARRAYA,N,ISIZE)     ! Rewrite in place
      CALL RDMWRITE (2,ARRAYB,N,ISIZE)
      END

C  Subroutine sets file to record M. Assume record sizes are fixed and
C  equal to ISIZE,which must be a multiple of 512.  IWA=0 is start of file.

      SUBROUTINE RDMWRITE (IUNIT,ARRAYX,M,MSIZE)
      DIMENSION ARRAYX(MSIZE)
      IWA = (M-1) * MSIZE            ! Start adrs = records x record size
      CALL IWAIT (IUNIT)             ! Unit must be idle for SETPOS to work
      CALL SETPOS (IUNIT,3,IWA)
      BUFFER OUT (IUNIT,1) (ARRAYX(1),ARRAYX(MSIZE))
      RETURN
      END

      SUBROUTINE RDMREAD (IUNIT,ARRAYX,M,ISIZE)
      DIMENSION ARRAYX(ISIZE)
      IWA = (M-1)*ISIZE
      CALL IWAIT (IUNIT)
      CALL SETPOS (IUNIT,3,IWA)
      BUFFER IN (IUNIT,1) (ARRAYX(1),ARRAYX(ISIZE))
      RETURN
      END

      SUBROUTINE IWAIT (NUNIT)
      X = UNIT(NUNIT)                  ! Waits for operation to complete
      END
```

9.3 NAMELIST STATEMENT (CFT77 EXTENSION)

The NAMELIST statement gives a name to a list of variables or arrays; the name can then be used in an I/O statement in place of the I/O list. Namelist I/O uses a standard format for input and output files, in the form *var=value*; for example X=1.0 . Figure 9-1 is an example showing Fortran code with namelist input and output.

```
NAMELIST/group/v[,v]...[[,]/group/v[,v]...]...
```

group Group name for the following list, cannot be used in any other way within the program unit.

v Variable name or array name. *v* cannot be a dummy argument, a pointee, or a variable in TASK COMMON.

In an I/O statement, the group name *group* is used in place of the I/O list but is entered in the position where the format specifier normally appears. A group name can be used in the following I/O statements (keywords explained in 7.7):

```
READ      (unit,group [,ERR=sn,END=sn])
WRITE     (unit,group [,ERR=sn])
READ      group
PRINT     group
PUNCH     group                      ! See E.4
```

Example:

```
NAMELIST/RECORD1/SIZE1,NUM1
READ(5,RECORD1)
WRITE(6,RECORD1)
```

The above READ statement would read the following input record:

```
&RECORD1  NUM1=25,SIZE1=1.234  &END
```

The above WRITE statement would write the following record:

```
&RECORD1  SIZE1=1.234  NUM1=25  &END
```

Every occurrence of a group name in any NAMELIST statement after the first occurrence is treated as a continuation of the first. Lists with the same group name are treated as a single group.

Program:

```
      PROGRAM EXAMPLE         ! Typical NAMELIST 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
      GOTO 10
      END
```

Input:

```
      Input data for program EXAMPLE
  Note that comment lines may be interspersed between complete groups
     $INPUT  $    Use values from DATA stmt if not in file
  $INPUT  LENGTH = 3.0,      ; a long wide case
        WIDTH = 3.    $
 &INPUT       DENSITY =  .5       &END
   &INPUT       ALLDONE = TRUE    &
/EOF
```

Output:

```
&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
```

Figure 9-1. Sample Program Using NAMELIST, with Input and Output

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.

9.3.1 NAMELIST INPUT

An input NAMELIST group record can consist of one or more physical records. Column 1 is ignored, except for a possible echo flag (CRAY X-MP only). In the first physical record, 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 as needed, with their assigned values. You may specify as many variables as you want and in any order. Use commas to separate items and to separate values for elements of the same array. Input items take the following forms.

```
variable=value

array=value[,value,]...

array(subscripts)=value[,value,]...
```

subscript An integer constant; multiple array values are assigned in storage order. Any value can be repeated by *n*value*, where *n* is the repetition count.

Example:

```
REAL X(100)
NAMELIST/RECORD1/I,X
READ(5,RECORD1)
```

The preceding code uses the following input record:

```
&RECORD1  I=10,X=50*0.,51.0,49*0.   &END
```

An input NAMELIST physical record can contain up to 152 characters. Your site analyst can change this value by altering values in the source code for the RNL routine, reassembling, and loading the new version with your program.

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.

9.3.1.1 NAMELIST input variables

NAMELIST input variables can be of type integer, 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 for assignment statements (see table 5-1), except that conversions between complex and double precision, or logical and any other type are not allowed.

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 and hexadecimal constants are considered to be Boolean. They are specified as follows:

- Octal constants are specified as *dddd...d*B or as O'*ddd...d*['], where each *d* is a digit from 0 through 7; each value can include a maximum of 22 digits.

- Hexadecimal constants are specified as Z'*hhh...h*['], where each *h* is a hexadecimal digit from A through F; each value can include a maximum of 16 digits.

If octal and hexadecimal values contain fewer than the maximum number of digits shown above, the values are right-justified in the input word.

Logical values are specified in one of the following ways:

.T[*string*]	.F[*string*]
T[*string*]	F[*string*]

string An optional string used for clarity. For example, .TRUE is more explicit than T . *string* cannot contain the following characters, which have special meanings in namelist I/O:

=	Replacement
$ or &	Delimiter
,	Separator
: or ;	Comment
()	Array name indicator

Complex constants are represented as follows:

(*real*,*imag*)

real and *imag* can be integer or floating-point constants.

9.3.1.2 NAMELIST input processing

Variables encountered in the NAMELIST record are redefined with the value specified; other variables in the group retain their current value.

The NAMELIST processor scans forward from the current position on the input file until it encounters a delimiter ($ or &) as the first nonblank character immediately followed by the group name.

If end-of-file or end-of-data is encountered before the group name is located, the job either aborts or branches to the END= address.

If the processor finds a NAMELIST record other than the one it is looking for, that record is skipped with an informative message to the logfile.

If the processor encounters an echo flag (E) in column 1 of any record, that record and all subsequent records processed by the current read are echoed to $OUT.

The job aborts or the ERR= branch is taken if one or more of the following conditions exists.

- The record contains a variable name that is not in a NAMELIST group.
- Punctuation is missing.
- The format of a constant is illegal.

9.3.1.3 User control subroutines

The following routines provide for control of the NAMELIST input defaults. The mode setting indicates the action to be taken.

CALL RNLSKIP(*mode*)	Determines action taken if NAMELIST sees a group name that is not the one being sought
mode > 0	Skips the record and issues a logfile message (default)
mode = 0	Skips the record
mode < 0	Aborts the job or goes to the optional ERR= branch
CALL RNLTYPE(*mode*)	Determines action taken if a type mismatch occurs across the equal sign
mode ≠ 0	Converts the constant to the type of the variable (default)
mode = 0	Aborts the job or goes to the optional ERR= branch
CALL RNLECHO(*unit*)	Specifies output unit for error message and echo lines
unit < 0	Specifies that error messages go to $OUT. Lines echoed because of an E in column 1 go to $OUT. (Default)
unit ≥ 0	Specifies that error messages go to *unit*. All input lines are echoed on *unit*, regardless of any echo flags present. (*unit*=6 or 101 imply $OUT.)

In the following user control subroutine argument lists, *char* is a character specified as 1L*x* or 1R*x*, and *mode* is a value which, if nonzero, adds the character to the set and which, if zero, removes the character from the set.

No checks are made to determine the 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 ;.)

9.3.2 NAMELIST OUTPUT

An output NAMELIST group record is written in the following general form:

```
& group  variable=value,..., array=value,...,value,...,&END
```

group, variable, and *array*
Names corresponding to 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.

The default line width is 133 characters unless (COS only) the unit is 102 ($PUNCH), in which case the default line width is 80 characters. NAMELIST output is readable as NAMELIST input.

9.3.2.1 User control subroutines

The following routines provide the user control of NAMELIST output format. *char* can be any ASCII character specified by 1L*x*, or 1R*x*. No checks are made to determine if *char* is reasonable, useful, or consistent with other characters. If the default characters are changed, use of the output line as NAMELIST input might not be possible.

CALL WNLLONG(*len*) — Sets the output line length to *len*. 8<*len*<161. If *len* is too short for an actual output line, the job aborts. *len*=-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

OPTIMIZATION 10

Run-time efficiency, a prime objective of the CFT77 compiler, consists of producing the most effective instruction sequence for each Fortran statement, and making full use of all Cray hardware capabilities and techniques to enhance execution speed. The compiler's capability for automatic optimizing of code is being constantly improved.

10.1 ANALYZING YOUR PROGRAM AND ITS PERFORMANCE

This subsection introduces Cray utilities to help you analyze your program structure, determine which parts of the program use the most execution time, and assess machine performance. These are described in detail in the Performance Utilities Reference Manuals, publication SR-2040 for UNICOS and publication SR-0146 for COS.

10.1.1 FTREF

FTREF statically analyzes your source code to show program structure and the use of common blocks. Usage examples are shown in 4.6.1. Additional information specific to multitasking can be generated by adding **-n** to the **ftref** command under UNICOS, or by adding MULTI to the FTREF control statement under COS.

10.1.2 FLOWTRACE

The Flowtrace feature monitors your program as it executes and prints statistics about subprogram calls and their timings. Usage examples of Flowtrace are shown in 1.4.4.1. You can also get statistics on specific areas of code, such as loops, by inserting calls to FLOWMARK, also discussed in 1.4.4.1. (FLOWMARK is available under UNICOS on CRAY-2 systems, and will be available under UNICOS 4.0 for CRAY X-MP systems.)

10.1.3 **prof** AND SPY

prof (under UNICOS†) and SPY (under COS) monitor program execution, but with a different emphasis than Flowtrace. The results are finer-grained than those of Flowtrace and contain almost no overhead from the utilities themselves; "spikes" in the output histogram clearly show where the most execution time is spent. However, these utilities do not provide Flowtrace information such as counts of subprogram calls, a call tree, or other information on the calling sequence.

The example below shows UNICOS commands for using **prof.** Before you follow this procedure, create file **profdir**, containing the directive **lib=/lib/libprof.a** . Example for program **hello:**

```
cft77 -e sg hello.f
segldr hello.o profdir      /* see directive above
a.out
prof > hello.prof
```

File **hello.prof** contains **prof** output; file **hello.l** contains generated code and source code listings.

In the **prof** output, a spike in the histogram indicates an address range where a large amount of execution time was spent. The address on the same line as the spike is the first address of the range. This address is part of a routine indicated on a previous line that shows only a name and an address. In the CAL listing for that routine, search the left column for the address that is shown in the Address column of the spike line in the **prof** output. On the CAL line with that address, the number in the right column is the number of a line in your source code listing; this part of your program caused the timing spike.

The following example shows COS JCL for using SPY; it is part of your JCL file in $IN, following your JOB and ACCOUNT statements. It is assumed that other required user files are also contained in $IN. The example is for program MYPROG, with SPY output in dataset $OUT.

```
...
CFT77,DEBUG.
SEGLDR,CMD='ABS=MYPROG'.
SPY,PREP.
MYPROG.
SPY,POST.
/EOF
```

Labels in SPY output are, where possible, based on labels of DO loops in your program. For example, 11B is the second address bucket for the DO 11 loop in a program unit.

† **prof** will be available on CRAY X-MP systems under UNICOS 4.0.

10.1.4 PERFTRACE (CRAY X-MP SYSTEMS ONLY)

Perftrace reports computer performance for each program unit in a program. Perftrace output gives information about computations, hold-issue conditions, memory use, or vectorization.

The example below shows UNICOS commands for using Perftrace. Before you follow this procedure, create file **perfdir**, containing the directive **lib=/lib/libperf.a**; it can also contain the directive **set=group:***n*, where *n* is either 2, 3, or 4. Without the **set** directive, Pertrace gives an execution summary. 2 reports on hold-issue conditions, 3 on memory references, and 4 on instruction and vector operation. Example for program **hello**:

```
cft77 -e f hello.f
segldr hello.o perfdir  /*Ignore messages*/
a.out > hello.perf
```

The following example shows COS JCL for using Perftrace; it is part of your JCL file in $IN, following your JOB and ACCOUNT statements. It is assumed that other required user files are also contained in $IN. Perftrace output is in dataset $OUT.

```
CFT77,ON=F.
SEGLDR,CMD='LIB=$PERFLIB;SET=GROUP:n',GO.
```

The underscored part in the SEGLDR command is optional. *n* has the same meaning as described in the previous UNICOS example.

10.2 MULTITASKING

Multitasking is the simultaneous use of different processors to increase the speed of execution. The subroutines for multitasking are all usable with CFT77. The Multitasking Programmer's Manuals, CRI publications SR-0222 (CRAY X-MP) and SN-2026 (CRAY-2) describe the use of these routines, which are called from CFT77 by the CALL statement.

10.3 VECTORIZATION

Vectorization is the process of changing an operation to a vector operation. An operation that can be vectorized is one that is in a loop and operates on successive array elements; this is replaced by an operation operating on vectors of array elements. A vector operation takes a vector, two vectors, or a vector and a scalar, and produces a vector of results. On a Cray computer, a vector operation produces a vector of up to 64 results. CFT77 vectorizes only when the compiler can determine that the program's meaning will not change.

The following terms are used in this subsection:

- A *branch* is a transfer of control caused by a GOTO, arithmetic IF, or alternate return.
- A *backward branch* is a GOTO or arithmetic IF statement that transfers control to a point in the program preceding that statement.
- A *constant increment variable* is a variable that is incremented by a constant amount on each iteration of a loop.
- An *invariant* value is a constant or simple expression referenced but not redefined within a loop. An *invariant expression* contains only invariant values.
- A *vector array reference* is an array element reference whose subscript expression is not invariant.
- A *recurrence* is a set of expressions in a loop in which computation of each expression requires the value of the expression from a previous iteration of the loop.

10.3.1 VECTORIZABLE LOOPS

A vectorizable loop is a DO loop; exceptions to this are described in 10.3.5. If any statement branches into a loop from outside it, the loop cannot be vectorized. If a loop contains a backward branch, the loop cannot be vectorized.

10.3.2 VECTORIZABLE STATEMENTS

To be vectorized, an arithmetic or logical assignment statement must have a vector array reference on the left side and must not involve a recurrence. Some particular types of recurrence can be vectorized; see 10.3.5.

Character assignment statements are not vectorized. IF statements can be vectorized if the IF condition is a vectorizable expression, defined later in this subsection. Other statements aren't vectorized. However, a vectorizable expression used as an argument in a CALL or WRITE statement can be vectorized if it is not part of a recurrence.

10.3.3 VECTORIZABLE EXPRESSIONS

An arithmetic or logical expression is vectorizable if it is in a vectorizable loop and is not part of a recurrence. Vectorizable expressions can include constants, invariant variables, and array element references. CFT77 does not vectorize external function references except VFUNCTION references.

An array element whose subscript is a linear function of a constant increment integer is vectorized using a vector load or store. An array element whose subscript is not a linear function of a constant increment variable, but is some other vectorizable expression, is vectorized using gather/scatter instructions if available. An array element with an invariant subscript expression is treated as a scalar.

Examples:

(1)
```
      REAL A(100), B(100), C(100), D(100), E(100)
      INTEGER INDEX(100)
      DO 10 I=1,100
         A(I) = 6.0 + I
         C(I) = 0.0
         D(INDEX(I)) = B(I) + E(1)
   10 CONTINUE
```

All the expressions in the above loop are vectorizable.

(2)
```
      DO 10 I=1,100
       A(INDEX(I)) = A(INDEX2(I))
   10 CONTINUE
```

The above loop will not be vectorized: CFT77 often has to assume that statements containing array elements with nonlinear subscripts involve recurrences and therefore cannot be vectorized.

10.3.4 LOOPS CONTAINING IFS

For an IF to be vectorized, the IF condition must be a vectorizable expression. Loops containing IF statements are compiled using either conditional vector merge hardware or gather-scatter and compress-index hardware.

Vectorization of a loop is not prevented by any of the following uses of IF statements within the loop:

- A block IF
- An arithmetic IF, including one that transfers control out of the loop
- A vectorizable IF controlling a GOTO that transfers out of the loop
- A logical IF whose conditional statement either is not a branch or branches forward within the same loop

10.3.5 RECURRENCES

A *recurrence* is a set of expressions in a loop in which computation of each expression requires the value of the expression from a previous iteration of the loop. Recurrences cannot normally be vectorized, because many iterations of a loop are calculated simultaneously, so values from previous iterations are often unavailable.

Example:

```
      REAL A, B(0:100), T, C
      DO 10 I=1,100
    5 A = A + 1
    6 B(I) = B(I-1)
    7 T = C
    8 C = T + 1
   10 CONTINUE
```

Statements 5 and 6 are recurrences, as is the pair of statements 7 and 8.

CFT77 detects recurrences, and does not vectorize an expression unless the expression is not part of a recurrence. (Exceptions are listed below.) When CFT77's recurrence analysis cannot determine whether a recurrence exists, it can take one of two courses of action. In some cases it assumes a recurrence and does not vectorize the expressions involved. In other cases, it decides that the recurrence exists only for certain array subscript values; it then generates code to compute the expression as scalar or vector, depending on the existence of a recurrence.

Recurrences that can be vectorized are as follows:

- First order linear recurrences (vector reductions). Some recurrences are reductions; that is, they apply a binary operation to all the elements of a linear array, producing a scalar result. Example:

  ```
        DO 10 I=1,100
        R = R + A(I)
     10 CONTINUE
  ```

  Reductions where the operation is multiply, add, divide, logical AND, logical OR, MAX or MIN can be vectorized.

- Recurrences with a large threshold. The threshold of a recurrence is the number of iterations that occur before a value is reused. Example:

  ```
        DO 10 I = 7, 100
        A(I) = A(I-6) + 1.0
     10 CONTINUE
  ```

  In the preceding code, the recurrence has a threshold of 6. A recurrence with a threshold greater than 64 can be vectorized on the Cray computer system. A recurrence with a threshold of k, $1<k<64$, can be vectorized with a vector length of k.

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 in the CFT77 character set are identified by the letter C in the fifth column. All other characters are members of the auxiliary character set. The letter A in the fifth column indicates those characters belonging to the ANSI FORTRAN character set.

Letters in parentheses following the descriptions in the sixth column indicate the following control character usage.

- CC - Communication control
- FE - Format effector
- IS - Information separator

Table A-1. Character Sets: ASCII, FORTRAN 77, CFT77

Character	Octal	Decimal	Hex	ANSI, Cray	Description
NUL	000	000	00		Null
SOH	001	001	01		Start of heading (CC)
STX	002	002	02		Start of text (CC)
ETX	003	003	03		End of text (CC)
EOT	004	004	04		End of transmission (CC)
ENQ	005	005	05		Enquiry (CC)
ACK	006	006	06		Acknowledge (CC)
BEL	007	007	07		Bell (audible signal)
BS	010	008	08		Backspace (FE)
HT	011	009	09	C	Horizontal tabulation (FE)
LF	012	010	0A		Line feed (FE)
VT	013	011	0B		Vertical tabulation (FE)
FF	014	012	0C		Form feed (FE)
CR	015	013	0D		Carriage return (FE)
SO	016	014	0E		Shift out
SI	017	015	0F		Shift in
DLE	020	016	10		Data link escape (CC)
DC1	021	017	11		Device control 1
DC2	022	018	12		Device control 2
DC3	023	019	13		Device control 3
DC4	024	020	14		Device control 4 (stop)
NAK	025	021	15		Negative acknowledge (CC)
SYN	026	022	16		Synchronous idle (CC)
ETB	027	023	17		End of transmission block (CC)
CAN	030	024	18		Cancel
EM	031	025	19		End of medium
SUB	032	026	1A		Substitute
ESC	033	027	1B		Escape
FS	034	028	1C		File separator (IS)
GS	035	029	1D		Group separator (IS)
RS	036	030	1E		Record separator (IS)
US	037	031	1F		Unit separator (IS)
space	040	032	20	A,C	Space (blank)
!	041	033	21	C	Exclamation point
"	042	034	22	C	Quotation marks
#	043	035	23		Number sign
$	044	036	24	A,C	Dollar sign (currency symbol)
%	045	037	25		Percent
&	046	038	26		Ampersand
'	047	039	27	A,C	Apostrophe (single quote)
(	050	040	28	A,C	Opening (left) parenthesis
)	051	041	29	A,C	Closing (right) parenthesis
*	052	042	2A	A,C	Asterisk
+	053	043	2B	A,C	Plus

Table A-1. Character Sets: ASCII, FORTRAN 77, CFT77 (continued)

Char-acter	Octal	Deci-mal	Hex	ANSI, Cray	Description
,	054	044	2C	A,C	Comma (cedilla)
-	055	045	2D	A,C	Minus (hyphen)
.	056	046	2E	A,C	Period (decimal point)
/	057	047	2F	A,C	Slant (slash, virgule)
0	060	048	30	A,C	Zero
1	061	049	31	A,C	One
2	062	050	32	A,C	Two
3	063	051	33	A,C	Three
4	064	052	34	A,C	Four
5	065	053	35	A,C	Five
6	066	054	36	A,C	Six
7	067	055	37	A,C	Seven
8	070	056	38	A,C	Eight
9	071	057	39	A,C	Nine
:	072	058	3A	A,C	Colon
;	073	059	3B		Semicolon
<	074	060	3C		Less than
=	075	061	3D	A,C	Equal
>	076	062	3E		Greater than
?	077	063	3F		Question mark
@	100	064	40		Commercial at-sign
A	101	065	41	A,C	Uppercase letter
B	102	066	42	A,C	Uppercase letter
C	103	067	43	A,C	Uppercase letter
D	104	068	44	A,C	Uppercase letter
E	105	069	45	A,C	Uppercase letter
F	106	070	46	A,C	Uppercase letter
G	107	071	47	A,C	Uppercase letter
H	110	072	48	A,C	Uppercase letter
I	111	073	49	A,C	Uppercase letter
J	112	074	4A	A,C	Uppercase letter
K	113	075	4B	A,C	Uppercase letter
L	114	076	4C	A,C	Uppercase letter
M	115	077	4D	A,C	Uppercase letter
N	116	078	4E	A,C	Uppercase letter
O	117	079	4F	A,C	Uppercase letter
P	120	080	50	A,C	Uppercase letter
Q	121	081	51	A,C	Uppercase letter
R	122	082	52	A,C	Uppercase letter
S	123	083	53	A,C	Uppercase letter
T	124	084	54	A,C	Uppercase letter
U	125	085	55	A,C	Uppercase letter
V	126	086	56	A,C	Uppercase letter
W	127	087	57	A,C	Uppercase letter

Table A-1. Character Sets: ASCII, FORTRAN 77, CFT77 (continued)

Character	Octal	Decimal	Hex	ANSI, Cray	Description
X	130	088	58	A,C	Uppercase letter
Y	131	089	59	A,C	Uppercase letter
Z	132	090	5A	A,C	Uppercase letter
[	133	091	5B		Opening (left) bracket
\	134	092	5C		Reverse slant (backslash)
]	135	093	5D		Closing (right) bracket
^	136	094	5E		Circumflex
_	137	095	5F	C	Underline
`	140	096	60		Grave accent
a	141	097	61	C	Lowercase letter
b	142	098	62	C	Lowercase letter
c	143	099	63	C	Lowercase letter
d	144	100	64	C	Lowercase letter
e	145	101	65	C	Lowercase letter
f	146	102	66	C	Lowercase letter
g	147	103	67	C	Lowercase letter
h	150	104	68	C	Lowercase letter
i	151	105	69	C	Lowercase letter
j	152	106	6A	C	Lowercase letter
k	153	107	6B	C	Lowercase letter
l	154	108	6C	C	Lowercase letter
m	155	109	6D	C	Lowercase letter
n	156	110	6E	C	Lowercase letter
o	157	111	6F	C	Lowercase letter
p	160	112	70	C	Lowercase letter
q	161	113	71	C	Lowercase letter
r	162	114	72	C	Lowercase letter
s	163	115	73	C	Lowercase letter
t	164	116	74	C	Lowercase letter
u	165	117	75	C	Lowercase letter
v	166	118	76	C	Lowercase letter
w	167	119	77	C	Lowercase letter
x	170	120	78	C	Lowercase letter
y	171	121	79	C	Lowercase letter
z	172	122	7A	C	Lowercase letter
{	173	123	7B		Opening (left) brace
|	174	124	7C		Vertical line
}	175	125	7D		Closing (right) brace
~	176	126	7E		Overline (tilde, general accent)
DEL	177	127	7F		Delete

INTRINSIC FUNCTIONS B

The tables in this section show the intrinsic functions available with CFT77. See 2.4.1 and 2.4.3. Conventions used in the tables are described below.

The rightmost column in each table includes letter codes to indicate conformance with the ANSI standard, the level of vectorization, and the kind of code generated. The codes use the following format:

First letter: ANSI standard
- A The function is specified in the ANSI FORTRAN 77 standard.
- C The function is a Cray extension to the standard.

Second letter: level of vectorization
- F Full vectorization
- N No vectorization

Third letter: what kind of code is generated
- E External
- I Inline

Unless noted otherwise, the first function name in each group of functions can be used as a *generic function name*. A generic function name can be used to call any of a group of related functions, so that one name can be used with arguments of different data types (see section 3). Groups of related functions are separated by horizontal lines, with the arguments and results shown for each specific function.

In the Definition column, y is the function's result, and the x values are function arguments separated by parentheses in the function call. Examples:

```
A = SQRT(B)            K = MOD(L,M)
```

Square brackets indicate truncation of a term: if *x* has a value of 5.67, [*x*] equals 5.

Data types shown in the Function and Argument columns are as follows:

I	Integer	P	Pointer
R	Real	B	Boolean
D	Double precision	CH	Character
C	Complex	L	Logical

Table B-1. General Arithmetic Functions

Purpose	Definition	Function Name,Type	Argument(s) No	Type	Range	Codes *
Truncation	$y=[x]$† No rounding. Also see INT, table B-2.	AINT R DINT D	1 1	R D	$\|x\|<2^{46}$ $\|x\|<2^{95}$	A F I A F E
Nearest whole number	$y=[x+.5]$† if $x\geq 0$ $y=[x-.5]$ if $x<0$ Also see INT, table B-2.	ANINT R DNINT D	1 1	R D	$\|x\|<2^{46}$ $\|x\|<2^{95}$	A F I A F E
Nearest integer	$y=[x+.5]$† if $x\geq 0$ $y=[x-.5]$ if $x<0$	NINT I IDNINT I	1 1	R D	$\|x\|<2^{46}$	A F I A F E
Absolute value	$y=\|x\|$ $y=(x_r{}^2+x_i{}^2)^{1/2}$	ABS R IABS I DABS D CABS†† R	1 1 1 1	R I D C	$\|x\|<$infinity	A F I A F I A F I A F E
Divide for remainder of x_1/x_2	$y=x_1-x_2[x_1/x_2]$	MOD I AMOD R DMOD D	2 2 2	I R D	†††	A F I A F I A F E
Transfer sign	$y=\|x_1\|$ if $x_2\geq 0$ or $y=-\|x_1\|$ if $x_2<0$	SIGN R ISIGN I DSIGN D	2 2 2	R I D	$\|x\|<$infinity	A F I A F I A F I

The first function name in each group can be used generically.

† [x] indicates that the fractional part of x is truncated.

†† $x=x_r+i\cdot x_i$

††† Argument ranges for MOD functions:

MOD: $|x_1|<2^{63}$ $0<|x_2|<2^{63}$ $2^{-63}<|x_1/x_2|<2^{63}$

AMOD: $|x_1|<2^{47}$ $0<|x_2|<2^{47}$ $2^{-47}<|x_1/x_2|<2^{47}$

DMOD: $|x_1|<2^{95}$ $0<|x_2|<2^{95}$ $2^{-95}<|x_1/x_2|<2^{95}$

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-1. General Arithmetic Functions (continued)

Purpose	Definition	Function Name, Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Positive difference	$y=x_1-x_2$ if $x_1>x_2$ $y=0$ if $x_1\le x_2$	DIM R IDIM I DDIM D	2 2 2	R I D	|x|<infinity	A F I A F I A F E
Double-precision product	$y=x_1 \cdot x_2$	DPROD D	2	R	|x|<infinity	A F I
Imaginary portion of complex value	$y=x_i$	AIMAG† R	1	C	|x|<infinity	A F I
Conjugate of complex value	$y=x_r-i \cdot x_i$	CONJG† C	1	C	|x|<infinity	A F I
Obtain random number††	y=r where r is the first or next in a series of random numbers ($0<y<1$).	RANF†† R	0			C F E
Obtain rndm seed	The current random number seed	RANGET I	1	I	|x|<infinity	C N E
Establish rndm seed	y=x	RANSET R	1	B	|x|<infinity	C N E

The first function name in each group can be used generically.

† $x=x_r+i \cdot x_i$

†† If RANF is called more than once in a loop, vectorization is inhibited, because the result differs from that produced by scalar code. You can force vectorization with CDIR$ IVDEP. This is further discussed at the end of this appendix.

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-2. Type Conversion Functions

Note: The functions listed below cannot be passed as arguments. Some functions in Table B-1 also change data types of arguments.

Purpose	Definition	Function Name,Type	Argument(s)			Codes *
			No	Type	Range	
Conversion to integer	Truncation toward zero (fraction lost)	INT I	1	C	$\|x_r\|<2^{46}$	A F I
		I	1	R	$\|x\|<2^{46}$	
		I	1	B†	$\|x\|<2^{63}$	
	Also see AINT, ANINT,	IFIX I	1	R,B†	$\|x\|<2^{46}$	A F I
	and NINT, table B-1.	IDINT I	1	D		A F I
Conversion to real	$y=x_r$††	REAL R	1	C	$\|x_r\|<$infin.	A F I
		R	1	R	$\|x\|<$infinity	
	y=x	R	1	I	$\|x\|<2^{46}$	
		FLOAT R	1	B†	$\|x\|<2^{46}$ or	A F I
	y=x Accuracy may be lost	SNGL R	1	D	2^{64} †††	A F I
Conversion to double-precision	$y=x_r$†† Accuracy may be lost.	DBLE D	1	C	$\|x_r\|<$infin.	A F I
		D	1	I	$\|x\|<2^{46}$	
	y=x Extra bit positions	D	1	R		
	are added to real	D	1	D	$\|x\|<$infinity	
	and Boolean values.	D	1	B†		
Conversion to complex	y=x	CMPLX C	1	C	$\|x\|<$infinity	A F I
		C	1,2	I	$\|x\|<2^{46}$	
	All two-argument uses can	C	1,2	B†		
	be either $y = x_1 + i \cdot x_2$	C	1,2	R	$\|x\|<$infinity	
	or $y = x_1 + i \cdot 0$	C	1,2	D		
Character to integer	ICHAR(x) returns position of character x in collating sequence	ICHAR I	1	CH	any legal character	A N E
Integer to character	CHAR(x) returns the xth character in collating sequence	CHAR CH	1	I	0-255	A N I
		CH	1	B		

† Type conversion routines assign appropriate types to Boolean arguments without shifting or manipulating the bit patterns they represent.

†† $x=x_r+i \cdot x_i$

††† When used with 46-bit or 64-bit integers, respectively.

Table B-3. Maximum/Minimum Functions†

Purpose	Definition	Function Name,Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Select maximum value	*Generic is MAX* y=The largest of all x	MAX0 I AMAX1 R DMAX1 D AMAX0 R MAX1 I	$64>n\geq 2$	I R D I R	$\lvert x\rvert$<infinity	A F I A F I A F I A F I A F I
Select minimum value	*Generic is MIN* y=The smallest of all x	MIN0 I AMIN1 R DMIN1 D AMIN0 R MIN1 I	$64>n\geq 2$	I R D I R	$\lvert x\rvert$<infinity	A F I A F I A F I A F I A F I

Table B-4. Character Functions†

Purpose	Definition	Function Name,Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Lex .GE.	$y=a_1\geq a_2$ in collating seq	LGE L	2	CH	any string	A N E
Lex .GT.	$y=a_1>a_2$ in collating seq	LGT L	2	CH	any string	A N E
Lex .LE.	$y=a_1\leq a_2$ in collating seq	LLE L	2	CH	any string	A N E
Lex .LT.	$y=a_1<a_2$ in collating seq	LLT L	2	CH	any string	A N E
Length of string	Number of characters in a character entity	LEN I	1	CH	any string	A N I
Index of a substrng	Returns the starting position of 2nd argument in 1st arg. 0 = not found	INDEX I	2	CH	any string	A N E

† Cannot be passed as arguments, except LEN and INDEX.

* A=ANSI standard, C=Cray extension; F=full vectorizing N=no vectorizing; E=external code, I=inline code.

Table B-5. Trigonometric Functions (Angles in radians)

Purpose	Definition	Function Name, Type		Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Sine	y=sin(x)	SIN	R	1	R	$\|x\|<2^{24}$	A F E
		DSIN	D	1	D	$\|x\|<2^{48}$	A F E
		CSIN†	C	1	C	$\|x_r\|<2^{24}$, $\|x_i\|<2^{13}\cdot\ln 2$	A F E
Cosine	y=cos(x)	COS	R	1	R	$\|x\|<2^{24}$	F E
		DCOS	D	1	D	$\|x\|<2^{48}$	F E
		CCOS†	C	1	C	$\|x_r\|<2^{24}$, $\|x_i\|<2^{13}\cdot\ln 2$	F E
Tangent	y=tan(x)	TAN	R	1	R	$\|x\|<2^{24}$	A F E
		DTAN	D	1	D		A F E
Cotangent	y=cot(x)	COT	R	1	R	$\|x\|<2^{24}$	C F E
		DCOT	D	1	D		C F E
Arcsine	y=arcsin(x)	ASIN	R	1	R	$\|x\|\leq 1$	A F E
		DASIN	D	1	D		A F E
Arccosine	y=arccos(x)	ACOS	R	1	R	$\|x\|\leq 1$	A F E
		DACOS	D	1	D		A F E
Arctangent	y=arctan(x)	ATAN	R	1	R		A F E
		DATAN	D	1	D	$\|x\|<$infinity	A F E
	$y=\arctan(x_1/x_2)$	ATAN2	R	2	R	$\|x_1\|<$infin.,	A F E
		DATAN2	D	2	D	$\|x_2\|\neq 0$	A F E
Hyperbolic sine	y=sinh(x)	SINH	R	1	R	$\|x\|<2^{13}\cdot\ln 2$	A F E
		DSINH	D	1	D		A F E

The first function name in each group can be used generically.

† $x=x_r+i\cdot x_i$

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-5. Trigonometric Functions (Angles in radians) (continued)

Purpose	Definition	Function Name, Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Hyperbolic cosine	$y=\cosh(x)$	COSH R DCOSH D	1 1	R D	$\|x\|<2^{13}\cdot\ln 2$	A F E A F E
Hyperbolic tangent	$y=\tanh(x)$	TANH R DTANH D	1 1	R D	$\|x\|<2^{13}\cdot\ln 2$	A F E A F E

Table B-6. Exponential Functions

Purpose	Definition	Function Name, Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Square root	$y=x^{1/2}$	SQRT R DSQRT D CSQRT† C	1 1 1	R D C	$0\leq x\leq$ infinity $x_r\geq 0$, $x_i<$ infinity	A F E A F E A F E
Exponent	$y=e^{x}$	EXP R DEXP D CEXP† C	1 1 1	R D C	$\|x\|<2^{13}\cdot\ln 2$ $\|x_r\|<2^{13}\cdot\ln 2$ $\|x_i\|<2^{24}$	A F E A F E A F E

The first function name in each group can be used generically.

Table B-7. Logarithmic Functions

Purpose	Definition	Function Name, Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Natural logarithm	$y=\ln(x)$ *Generic is LOG; cannot be an argument.*	ALOG R DLOG D CLOG† C	1 1 1	R D C	$0<x<$ infinity	A F E A F E A F E
Common logarithm	$y=\log(x)$ *Generic is LOG10. Cannot be an arg.*	ALOG10 R DLOG10 D	1 1	R D	$0<x<$ infinity	A F E A F E

† $x=x_r+i\cdot x_i$

Table B-8. Boolean and Logical Functions

Purpose	Definition	Function Name, Type	Argument(s)			Codes *
			No	Type	Range	
Logical product †	Logical product (AND) of x_1 and x_2	AND B L	2 2	B,I, R,P L		C F I
Logical sum †	Logical sum (OR) of x_1 and x_2	OR B L	2 2	B,I, R,P L		C F I
Logical difference (not equiv.) †	Logical difference (XOR or NEQV) of x_1 and x_2	XOR, NEQV B L	2 2	B,I, R,P L		C F I
Equivalence †	Equivalence ($\overline{XOR}$) of x_1 and x_2	EQV B L	2 2	B,I, R,P L		C F I
Complement †	Logical complement of x_1	COMPL B L	1 2	B,I, R,P L		C F I
Mask	x=number of one-bits to be left-justified if $0 \le x \le 63$. (128-x)=number of 1-bits to be right-justified if $64 \le x \le 128$	MASK B	1	I		C F I

† With logical operands, the function is a logical operation with a single logical result. With Boolean operands, the function is a masking operation; each bit in the result represents a separate Boolean operation on the corresponding bit in the operands. These types are discussed in 3.6 and 3.8; the operations are discussed in 5.4 and 5.5.

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-8. Boolean and Logical Functions (continued)

Purpose	Definition	Function Name, Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Circular shift	x_1 shifts left x_2 positions; leftmost positions replace vacated positions	SHIFT B	2	x_1: B,I, R,P x_2:I	$0 \le x_2 < 64$	C F I
Logical shift	x_1 shifts left x_2 positions; leftmost positions lost; rightmost positions set to zero	SHIFTL B	2	x_1: B,I, R,P x_2:I	$0 \le x_2 < 64$	C F I
	x_1 shifts right x_2 positions; rightmost positions lost; leftmost positions set to zero	SHIFTR B	2	x_1: B,I, R,P x_2:I		C F I
Leading zeros	Tallies number of leading zeros in x	LEADZ I	1	B,I, R,P		C F I
Population count	Tallies number of ones in x	POPCNT I	1	B,I, R,P		C F I
Population parity count	0, if x has an even number of ones; 1, if x has odd number of ones	POPPAR I	1	B,I, R,P		C F I
Conditionl Scalar Merge	Bit-by-bit selective merge: (x_1 AND x_3) OR (x_2 AND NOT x_3). See the end of this appendix.	CSMG B	3	B,I, R,P		C F I

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-9. Time and Date Functions

Purpose	Definition	Function Name, Type	Argument(s) No	Type	Range	Codes *
Real-time clock	Low order 46 bits of clock register expressed as floating point integer	RTC R	0			C N I
	Current clock register content	IRTC I	0			C N I
Time	Current time in ASCII code (hh:mm:ss)	CLOCK† B	0			C N E
Julian date	Current Julian date in ASCII code (yyddd)	JDATE† B	0			C N E
Date	Current date in ASCII code (mm/dd/yy)	DATE† B	0			C N E

† These procedures can be called as subroutines also. An integer or real argument is passed to return the result.

Table B-10. Miscellaneous Functions

Purpose	Definition	Function Name, Type	Argument(s) No	Type	Range	Codes *
Location	Returns memory address of specified variable or array	LOC P	1	B,I, R,L, C,D, P		C N I
Number of arguments	Tallies number of arguments used to call a subprogram	NUMARG I	0			C N I

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Table B-11. Conditional Vector Merge Functions

Tests for	Definition	Function Name	Function Type	Argument(s) No	Argument(s) Type	Argument(s) Range	Codes *
Positive or zero	x_1 returned if $x_3 \geq 0$ x_2 returned if $x_3 < 0$	CVMGP	B	3	B,I R,P		C F I
Minus (negative)	x_1 returned if $x_3 < 0$ x_2 returned if $x_3 \geq 0$	CVMGM	L	3	x_1, x_2: L		
Zero	x_1 returned if $x_3 = 0$ x_2 returned if $x_3 \neq 0$	CVMGZ			x_3: I,R B,P		
Non-zero	x_1 returned if $x_3 \neq 0$ x_2 returned if $x_3 = 0$	CVMGN					
True	x_1 returned if x_3 is true x_2 returned if x_3 is false	CVMGT	B	3	x_1,x_2: B,I, R,P x_3:L		C F I
			L	3	L		

* A=ANSI standard, C=Cray extension; F=full vectorizing, N=no vectorizing; E=external code, I=inline code.

Conditional Vector Merge (CVMG functions)

The conditional vector-merge instructions shown in the above table are used when an IF statement involving arrays prevents vectorization of a loop. Both CFT and CFT77 can now vectorize almost all such loops, but these functions are used in older codes. Scalar arguments can also be used with these functions. CVMG functions cannot be passed as arguments.

For operands other than logical type, the vector merge functions are not generic with respect to data typing. They accept arithmetic values of different types but interpret them as Boolean type (that is, as bit patterns). The returned function value is also Boolean. Therefore the function reference CVMGT(1.0,2.0,.TRUE.) is not type real (with a value of 1.0), but a Boolean value that acts the same as 1.0 in a floating-point operation.

A problem arises if you assume that the function reference is type real and use it in an expression or assignment that causes automatic type conversion. For example, if you use the function reference in a context where an integer is needed, the result is invalid because 1.0 and 1 have different bit patterns (see G.1 and G.2).

Because CVMG function values are Boolean, a binary operation involving two CVMG functions uses integer arithmetic. This can give unexpected results in assignments such as the following, where A, B, C, and D are real:

```
 X = CVMGT(A,B,LOGIC1) +       ! Integer arithmetic, invalid results
&    CVMGT(C,D,LOGIC2)
```

However, when used in an expression with another operand, a CVMG function value takes on the type of the other operand, without any explicit type conversion. For example, the following expression uses real arithmetic:

```
X = 1.0 + CVMGT(2.0,3.0,LEXP)    ! Valid because types agree
```

Following are suggestions for preventing bugs in the use of these functions:

- Use only one CVMG function in a given expression.

- If you use more than one CVMG function in an expression, use explicit type conversion; this does not generate any additional code. For example:

```
 X = REAL (CVMGT (1.0,2.0,LTEST)) +
&    REAL (CVMGT (X,Y,LTEST2))
```

- Make sure that the assignment type matches the function argument. For example:

```
      X = CVMGT (1.0,2.0,LTEST)

or    X = IFIX (CVMGT (1,2,LTEST))  ! Uses auto. type conversion

not   X = CVMGT (1,2,LTEST)         ! Type mismatch
```

- Never use mixed typing in the first two arguments of a CVMGT function. For example:

```
CVMGT (1,2.0,LTEST)     ! DON'T DO THIS!!
```

CSMG (Conditional Scalar Merge Function)

CSMG(x,y,z) equals (x .AND. z) .OR. (y .AND. .NOT.z)

CSMG merges x and y, controlled by the bit mask in z. When a 1 bit appears in z, the corresponding bit of x is taken; when a 0 bit appears in z, the corresponding bit of y is taken.

CSMG compiles to very efficient inline code because the Cray hardware includes a scalar merge instruction that is generated for CSMG. Because CSMG uses Cray-specific hardware, it should not normally be used where portability is important.

Some typical uses of CSMG follow.

To select one of the arguments, for any x and y:

CSMG(x,y,MASK(64)) equals CSMG(x,y,-1) equals x.

CSMG (x,y,0) equals y.

To replace bit fields:

```
  CSMG( 'ABCDEFGH'H, '12345678'H, X'0000FFFFFF0000FF' ) = '12CDE67H'H

  INTEGER EXPONENT, EXPMASK
  PARAMETER( EXPMASK = X'7FFF000000000000' )
  EXPONENT(X) = SHIFTR( X.AND.EXPMASK, 48 )  ! Statement function
  DIVBY2(X) =                                ! Statement function
 &   CSMG(  SHIFTL(EXPONENT(X)-1,48),
 &          X,
 &          EXPMASK )
```

Then, for example, DIVBY2(-28.0) = -14.0.

RANF (Get random number)

The use of the RANF function differs between CFT77 and CFT. CFT77 inhibits vectorization of a loop containing more than one RANF call, whereas CFT does not. This restriction is imposed because the sequence of function calls differs between vector and scalar code. Since each call gives (by definition) an unpredictable value, a different sequence gives different results.

Example:

```
      DO 10, I=1,64
   10 A(I) = RANF() + RANF()    ! CFT77 does not vectorize this; CFT does.
```

The above calls to RANF affect the result. If RANF() is called repeatedly, it returns a series of values x_1, x_2, x_3.... Using these values, the above loop is processed as follows:

Scalar Code:	Vector Code:
$A(1) = x_1 + x_2$	$A(1) = x_1 + x_{65}$
$A(2) = x_1 + x_4$	$A(2) = x_1 + x_{66}$
...	...

CFT77 follows a policy that vector and scalar code should always give the same results. Because the above results differ, CFT77 inhibits vectorization. Nevertheless, for many practical purposes, a change in the order of random numbers is insignificant. Therefore, to allow vectorization, you may wish to insert CDIR$ IVDEP before a loop containing more than one call to RANF.

POWERS AND CONSTANTS

C

Table C-1. Miscellaneous Constants

Constant	Decimal	Octal
1 Angstrom unit	10^{-8} cm	0 37746 5274616704302141
Avogadro's number	$6.0247 \pm .0002 \times 10^{23}$ mole^{-1} (physical scale)	0 40117 7762375551021105
Degree	0.01745 32925 19943 radians	0 37773 4357506504512347
e	2.71828 18284 59045 23536	0 40002 5337412426121273
K (constant of gravitation)	6.670×10^{-8} (The attraction in dynes between two gram masses one centimeter apart)	0 37751 4363626636225066
1 Knot	101.3 ft/min. 1.689 ft/sec. 1.152 mi/hr.	0 40007 6251463146314631 0 40001 6603044672274324 0 40001 4467227432477371
ln 2	0.69314 71805 59945 3	0 40000 5427102775750717
ln 10	2.30258 50929 94045 68402	0 40002 4465661567325250
$\log_{10}2$	0.30102 99956 63981	0 37777 4642023241175717
$\log_{10}e$	0.43429 44819 03251 82765	0 37777 6745573052233450
$\log_{10}\log_{10}e$	9.63778 43113 00537 - 10	1 37777 5627212554417747
$\log_{10}$ (pi)	0.49714 98726 94133 85435	0 37777 7750515546156435
$\log_{10}$ (5)	0.69897 00043 36019	0 40000 5456766257301030
1 micron	10^{-4} cm	0 37763 6433342726161031
Planck's constant	$6.6254 \pm .0002 \times 10^{-27}$ erg sec	0 37652 4063530471550577

Table C-1. Miscellaneous Constants (continued)

Constant	Decimal	Octal
pi	3.14159 26535 89793 23846 26433 83279 50	0 40002 6220773250420550
pi/180	0.01745 32925 19943 29576 92369 07684 9	0 37773 4357506504512351
$(pi/2)^2$	2.46740 11002 72339 6	0 40002 4736474623371056
$(pi/2)^3$	3.87578 45850 37477 4	0 40002 7600633262342353
$(pi/2)^4$	6.08806 81896 25152 0	0 40003 6055056430244101
$(pi/2)^5$	9.56311 51495 40044 9	0 40004 4620120501771201
$(pi/2)^6$	15.02170 61496 14130 7	0 40004 7405435043025557
$(pi/2)^7$	23.59604 08420 06186 2	0 40005 5714226103715771
$(pi/2)^8$	37.06457 24815 25675 7	0 40006 4504103722360577
$(pi/2)^9$	58.22089 71356 37125 9	0 40006 7216106266752535
$(pi/2)^{10}$	91.45317 13633 62315 3	0 40007 5556400604731077
$(pi/2)^{11}$	143.65430 56513 1374 95	0 40010 4372360044636773
$(pi/2)^{12}$	225.65165 56453 50	0 40010 7032332271702410
$(pi/2)^{13}$	354.45279 18229 1051 47	0 40011 5423476505215646
$(pi/2)^{14}$	556.77314 34176 24	0 40012 4263057313503564
1 radian	57.29577 95131 degrees	0 40006 7122734064617135
Velocity of light in a vacuum (Birge, 1941)	$2.99776x10^{10}$ cm/sec $9.83514x10^8$ ft/sec 186272 mi/sec	0 40043 6765467400000000 0 40036 7247635620000000 0 40022 5536400000000000

Powers of Two

Decimal	n	Octal	n	Hexadecimal
2	1	2	1	2
4	2	4	2	4
8	3	10	3	8
16	4	20	4	10
32	5	40	5	20
64	6	100	6	40
128	7	200	7	80
256	8	400	8	100
512	9	1000	9	200
1024	10	2000	10	400
2048	11	4000	11	800
4096	12	10000	12	1000
8192	13	20000	13	2000
16384	14	40000	14	4000
32768	15	1 00000	15	8000
65536	16	2 00000	16	10000
1 31072	17	4 00000	17	20000
2 62144	18	10 00000	18	40000
5 24288	19	20 00000	19	80000
10 48576	20	40 00000	20	1 00000
20 97152	21	100 00000	21	2 00000
41 94304	22	200 00000	22	4 00000
83 88608	23	400 00000	23	8 00000
167 77216	24	1000 00000	24	10 00000
335 54432	25	2000 00000	25	20 00000
671 08864	26	4000 00000	26	40 00000
1342 17728	27	10000 00000	27	80 00000
2684 35456	28	20000 00000	28	100 00000
5368 70912	29	40000 00000	29	200 00000
10737 41824	30	1 00000 00000	30	400 00000
21474 83648	31	2 00000 00000	31	800 00000
42949 67296	32	4 00000 00000	32	1000 00000
85899 34592	33	10 00000 00000	33	2000 00000
1 71798 69184	34	20 00000 00000	34	4000 00000
3 43597 38368	35	40 00000 00000	35	8000 00000
6 87194 76736	36	100 00000 00000	36	10000 00000
13 74389 53472	37	200 00000 00000	37	20000 00000
27 48779 06944	38	400 00000 00000	38	40000 00000
54 97558 13888	39	1000 00000 00000	39	80000 00000
109 95116 27776	40	2000 00000 00000	40	1 00000 00000
219 90232 55552	41	4000 00000 00000	41	2 00000 00000
439 80465 11104	42	10000 00000 00000	42	4 00000 00000
879 60930 22208	43	20000 00000 00000	43	8 00000 00000
1759 21860 44416	44	40000 00000 00000	44	10 00000 00000
3518 43720 88832	45	1 00000 00000 00000	45	20 00000 00000
7036 87441 77664	46	2 00000 00000 00000	46	40 00000 00000
14073 74883 55328	47	4 00000 00000 00000	47	80 00000 00000

28147 49767 10656	48	10 00000 00000 00000	48	100 00000 00000
56294 99534 21312	49	20 00000 00000 00000	49	200 00000 00000
1 12589 99068 42624	50	40 00000 00000 00000	50	400 00000 00000
2 25179 98136 85248	51	100 00000 00000 00000	51	800 00000 00000
4 50359 96273 70496	52	200 00000 00000 00000	52	1000 00000 00000
9 00719 92547 40992	53	400 00000 00000 00000	53	2000 00000 00000
18 01439 85094 81984	54	1000 00000 00000 00000	54	4000 00000 00000
36 02879 70189 63968	55	2000 00000 00000 00000	55	8000 00000 00000
72 05759 40379 27936	56	4000 00000 00000 00000	56	10000 00000 00000
144 11518 80758 55872	57	10000 00000 00000 00000	57	20000 00000 00000
288 23037 61517 11744	58	20000 00000 00000 00000	58	40000 00000 00000
576 46075 23034 23488	59	40000 00000 00000 00000	59	80000 00000 00000
1152 92150 46068 46976	60	100000 00000 00000 00000	60	100000 00000 00000
2305 84300 92136 93952	61	200000 00000 00000 00000	61	200000 00000 00000
4611 68601 84273 87904	62	400000 00000 00000 00000	62	400000 00000 00000
9223 37203 68547 75808	63	1000000 00000 00000 00000	63	800000 00000 00000

pi^2 = 9.86960 44010 89358 61883 43909 9988
2 x pi^2 = 19.73920 88021 78717 23766 87819 9976
3 x pi^2 = 29.60881 32032 68075 85680 31729 9964
4 x pi^2 = 39.47841 76043 57434 47533 75639 9952
5 x pi^2 = 49.34802 20054 46793 09417 19549 9940
6 x pi^2 = 59.21762 64065 36151 71300 63459 9928
7 x pi^2 = 69.08723 08076 25510 33184 07364 9916
8 x pi^2 = 78.95683 52087 14868 95067 51279 9904
9 x pi^2 = 88.82643 96098 04227 56950 95189 9892

$2^{0.5}$ = 1.41421 35623 73095 04880 1688
$1 + 2^{0.5}$ = 2.41421 35623 73095 04880 1688
$(1 + 2^{0.5})^2$ = 5.82842 71247 4618
$(1 + 2^{0.5})^4$ = 33.97056 27484 7708
$(1 + 2^{0.5})^6$ = 197.99494 93661 1630
$(1 + 2^{0.5})^8$ = 1153.99913 34482 2072
$(1 + 2^{0.5})^{10}$ = 6725.99985 13232 0802
$(1 + 2^{0.5})^{12}$ = 39201.99997 44910 2740
$(1 + 2^{0.5})^{14}$ = 228485.99999 56229 5638
$(1 + 2^{0.5})^{16}$ = 1331713.99999 92467 11
$(1 + 2^{0.5})^{18}$ = 7761797.99999 98847 51

In radians:

Sin .5 = 0.47942 55386 04203
Cos .5 = 0.87758 25618 90373
Tan .5 = 0.54630 24898 43790
Sin 1 = 0.84147 09848 07896
Cos 1 = 0.54030 23058 68140
Tan 1 = 1.55740 77246 5490
Sin 1.5 = 0.99749 49866 04054
Cos 1.5 = 0.07073 72016 67708
Tan 1.5 = 14.10141 99471 707

MEMORY MANAGEMENT D

Many programs contain a memory allocation scheme that expands an array in a common block located in Central Memory at the end of the program. This practice of expanding blank common or expanding a dynamic common block (sometimes referred to as overindexing) works under COS. Expanding blank common blocks under UNICOS, however, causes conflicts between user management of memory and the dynamic memory requirements of UNICOS libraries. CRI recommends that you modify programs rather than expand blank common, particularly when migrating.

Figures D-1 and D-2 show the structure of UNICOS and COS in relation to expanding blank common. In both figures, the user area includes code, data, and common blocks.

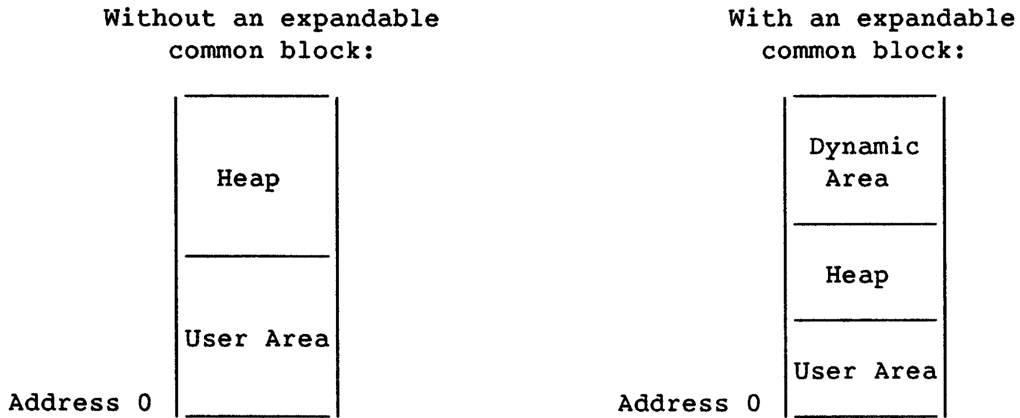


Figure D-1. Memory Use under UNICOS

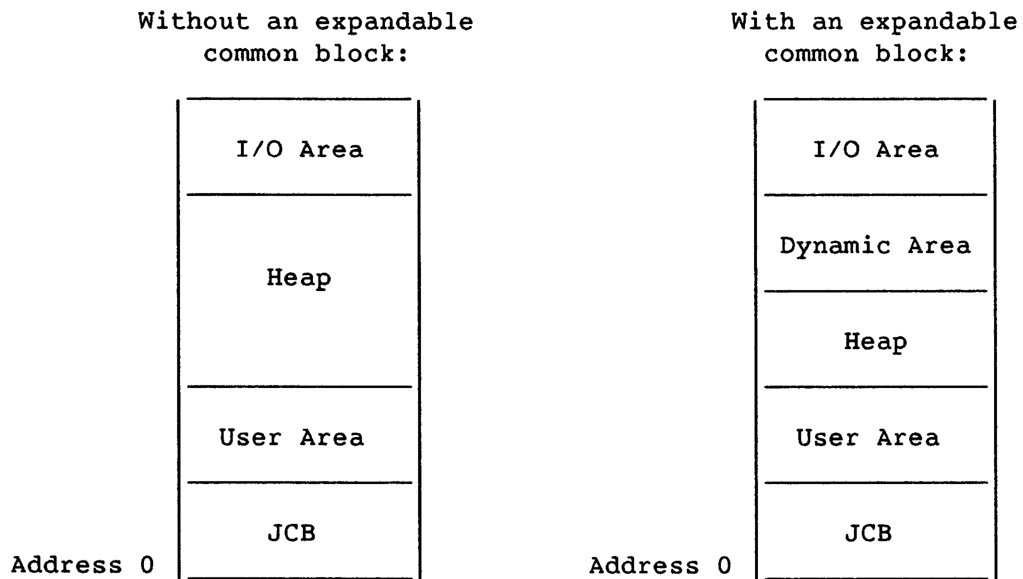


Figure D-2. Memory Use under COS

There are two ways to change your code. The first method, outlined in D.1, is the preferred method.

D.1 CHANGING YOUR CODE: RECOMMENDED METHOD

CRI recommends that you change your program using the following two-step process. This method will work under both COS and UNICOS.

Step 1 - For arrays that expand in a common block, define pointers that will point to the first address in each array.

Step 2 - Change any calls to memory to calls to library routine HPALLOC.

Original Code:

```
      PROGRAM TEST
      COMMON/WORK/IPTR
      ...
      ...
      ...
      CALL MEMORY ('UC',100000)
      ...
      DO 10 I = 1,100000
        X(I) = RANF( )
   10 CONTINUE
      ...
```

Converted Code (After Steps 1 and 2):

```
      PROGRAM TEST
      COMMON/WORK/IPTR
      ...
C     STEP 1:
      POINTER (IPTR,X(1))
      ...
C     STEP 2:
      CALL HPALLOC (IPTR,100000,ERRCODE,ABORT))
      ...
      DO 10 I = 1,100000
        X(I) = RANF( )
   10 CONTINUE
      ...
```

D.2 CHANGING YOUR CODE: ALTERNATIVE METHOD

If possible, use the conversion method outlined in D.1. You can also use the following two-step method for moving COS code to a CRAY X-MP UNICOS system, but not to a CRAY-2 UNICOS system.

Step 1:

Modify the program by changing calls to MEMORY to calls to SBRK. SBRK calls system routine **sbrk** to instruct UNICOS to allocate space for the planned expansion. (See **brk**(2) in the UNICOS System Calls Reference Manual, publication SR-2012.)

COS Example:

```
      PROGRAM TEST
      ...
      COMMON/WORK/X(1)
      ...
      CALL MEMORY ('UC',100000)
      ...
      DO 10 I = 1,100000
        X(I) = RANF( )
   10 CONTINUE
      ...
      END
```

UNICOS Example:

```
      PROGRAM TEST
      ...
      COMMON/WORK/X(1)
      ...
      CALL SBRK (100000)
      ...
      DO 10 I = 1,100000
        X(I) = RANF( )
   10 CONTINUE
      ...
      END
```

Step 2:

When building the program, specify the following SEGLDR directives:

DYNAMIC=*comblk* or DYNAMIC=//

comblk is the common block to be expanded. // specifies blank common as dynamic. This directive is also required for expanding under COS.

HEAP=*init*

init is the initial size of the heap, which contains I/O buffers and calling sequence information along with any user-requested heap areas. Since the heap cannot be expanded when a dynamic common block exists, you must make sure that the heap is initially large enough to accommodate your program's needs concerning buffering and the calling sequence requirements. Generally, when the heap cannot expand, the value of *init* must be at least 5000, but larger values may be required.

For more information on SEGLDR, see the Segment Loader (SEGLDR) Reference Manual, CRI publication SR-0066.

OUTMODED FEATURES — E

This appendix describes obsolete or nonstandard FORTRAN features supported by CFT77 that have been replaced by alternatives that enhance the portability of CFT77 programs. The outmoded features and their preferred alternatives are as follows.

Obsolete Feature	Preferred Alternative
Hollerith data	Character data
ENCODE and DECODE	Internal files
Asterisk format descriptor	Quotation mark format descriptor
[-*b*]X format descriptor	TL format descriptor
R descriptor and A used for noncharacter data	Character type and other conventional matchings of data with descriptors
EOF, IEOF, and IOSTAT functions	End-of-file specifier (END=) or status specifier (IOSTAT=)
PUNCH statement	WRITE statement
DATA statement with declaratives	DATA statement after other declaratives
DATA statement *nlist/clist* logical/Hollerith correspondence	*nlist/clist* both logical or both character
DOUBLE type statement	DOUBLE PRECISION type statement
DOUBLE FUNCTION statement	DOUBLE PRECISION FUNCTION statement
Type statements with **n*	Standard type statements
Two-branch arithmetic IF	Arithmetic IF or block IF
Indirect logical IF	Logical IF

E.1 HOLLERITH TYPE

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.

E.1.1 HOLLERITH CONSTANTS

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 that 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 between delimiting apostrophes or quotation marks are interpreted and counted as a single apostrophe within the sequence. Thus the sequence DON'T USE "*" would be specified with apostrophe delimiters as 'DON''T USE "*"'H, and with quotation mark delimiters as "DON'T USE ""*"""H.

Each character of a Hollerith constant is represented internally by an 8-bit code (see appendix A) with up to eight such codes in a single 64-bit Cray word. The codes corresponding to character positions 1 through 8 of a Hollerith constant are sequentially represented from left to right in a Cray word. The unused portion of the word is to the right and contains up to seven blank character codes (20_{16}).

When the number of characters in a character sequence is fewer than eight, the Cray word used can contain up to seven null character codes (00_{16}). Null codes can be produced by substituting the letter L for the letter H in the Hollerith forms described above.

When fewer than 8 characters appear in a Hollerith constant, the unused portion of a Cray word can contain up to seven null character codes at the left. The null codes can be produced by substituting the letter R for the letter H in the first form of Hollerith constant expression or by following 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 word containing the constant:

Hollerith Constant (bit position)	Internal Representation (64-bit Cray word) (0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63)
6HCRAY-2	C	R	A	Y	-	2	(20_{16})	(20_{16})
'CRAY-2'H	C	R	A	Y	-	2	(20_{16})	(20_{16})
"CRAY-2"H	C	R	A	Y	-	2	(20_{16})	(20_{16})
6LCRAY-2	C	R	A	Y	-	2	(00)	(00)
'CRAY-2'L	C	R	A	Y	-	2	(00)	(00)
"CRAY-2"L	C	R	A	Y	-	2	(00)	(00)
6RCRAY-2	(00)	(00)	C	R	A	Y	-	2
'CRAY-2'R	(00)	(00)	C	R	A	Y	-	2
"CRAY-2"R	(00)	(00)	C	R	A	Y	-	2

A Hollerith constant is limited to 8 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 eight characters. An all-zero computer word follows the last word containing a Hollerith constant specified as an actual argument in an argument list.

A character constant of 8 or fewer characters is used as a Hollerith constant in situations where a character constant is illegal and a Hollerith constant is legal. See 5.2.3.

NAMELIST Hollerith constants are specified by the following forms.

*n*H...
*n*L...
*n*R...

```
      H
'...' L
      R

      H
"..." L
      R
```

If the R form is used, the string must contain 8 or less characters. Within the ' or " delimited format, a ' or " is specified as '' or "", respectively.

E.1.2 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
- The name of a variable containing a Hollerith datum
- The name of an array element containing a Hollerith datum
- The name of a function that provides a Hollerith datum when referenced

A Hollerith constant comprising a Hollerith expression is limited to 8 characters.

The data type of the name referencing a variable or array element containing a Hollerith datum can affect its evaluation during program execution. A variable or array element of type integer or real contains 8 Hollerith characters. Hollerith characters can appear only in variables, array elements, or DATA statements of type integer, real, or logical. A variable or array element of type logical containing Hollerith characters must first be initialized in a DATA statement.

A Hollerith datum 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 integer.

E.1.3 HOLLERITH RELATIONAL EXPRESSIONS

Used with a relational operator, the Hollerith expression e_1 is less than e_2 if its value precedes the value of $e(2)$ in the collating sequence and is 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.LOCK
```

E.2 FORMATTED DATA ASSIGNMENT

Formatted data assignment operations define entities by transferring data between input/output list items and internal records. The preferred method for achieving this is now the use of internal files (see 7.4).

Like other assignment statements, formatted data assignment statements perform only 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 (*n,f,dent*)[*elist*]
DECODE (*n,f,sent*)[*dlist*]

n Number of characters to be processed, specified by a nonzero integer expression not to exceed 152

f Format identifier, except for an asterisk

dent Symbolic name of a destination variable, array element, or array where the *n* characters of *elist* are packed (8 per word) by ENCODE

sent Symbolic name of the source variable, array element, or array where characters are unpacked and stored into *dlist* by DECODE

elist, dlist
Lists specified the same as for formatted input/output statements. *elist* is the list written to the destination entity; *dlist* is the list receiving the source entity.

E.2.1 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 in a variable, array element or array identified by *dent*.

If *n* is not an integer multiple of eight, the last word in each record is padded with spaces to a word boundary. In effect, *n* is rounded up to be a multiple of eight.

Example:

```
    elist:     array ZD(5):  ZD(1) = 'THISbbbb'
                             ZD(2) = 'MUSTbbbb'
                             ZD(3) = 'HAVEbbbb'
                             ZD(4) = 'FOURbbbb'
                             ZD(5) = 'CHARbbbb'

        f:     FORMAT (5A4)
        n =    20
     dent:     array ZE(3)
```

The sequence

```
      ENCODE (20,1,ZE)ZD
   1  FORMAT (5A4)
```

produces

```
    dent =     ZE(1) = 'THISMUST'
               ZE(2) = 'HAVEFOUR'
               ZE(3) = 'CHARbbbb'
```

E.2.2 DECODE STATEMENT

The DECODE statement processes a sequence of *n* characters (packed eight per word) contained in the variable, array element, or array identified by *sent* under control of the format specification identified by *f*. The resulting values define the input list items specified in *dlist*.

If *n* is not an integer multiple of eight and the DECODE format calls for more than one DECODE record, the second and all subsequent DECODE records begin on a word boundary. In effect, *n* is rounded up to be a multiple of eight.

Example:

sent:	ZE:	ZE(1) = 'WHILETHI'
		ZE(2) = 'SbHASbbF'
		ZE(3) = 'IVEbbbbb'
n: =	20	
f:	FORMAT (5A5)	

The sequence

```
    DECODE (20,2,ZE)ZD
  2 FORMAT (4A5)
```

produces

dlist =	ZD(1) = 'WHILEbbb'
	ZD(2) = 'THISbbbb'
	ZD(3) = 'HASbbbbb'
	ZD(4) = 'FIVEbbbb'

E.3 FORMAT DESCRIPTORS

This subsection shows obsolete format descriptors and outmoded uses of current descriptors.

E.3.1 ASTERISK DELIMITERS

The asterisk is used to delimit a literal character constant. It has been replaced by the single quotation mark (apostrophe).

$*h_1h_2...h_n*$

*	Delimiter for a literal character string
h_i	Any ASCII character indicated by a C in appendix A (that is, capable of internal representation)

Example:

```
*AN ASTERISK FORMAT DESCRIPTOR*
```

E.3.2 NEGATIVE-VALUED X DESCRIPTOR

A negative value can be used with the X descriptor to indicate a move to the left. This has been replaced by the TL descriptor.

[-*b*]X

b Any nonzero, unsigned integer constant

X Indicates a move of as many positions as indicated by *b*

Example:

-55X (Moves current position 55 spaces to left.)

E.3.3 A AND R DESCRIPTORS FOR NONCHARACTER TYPES

The Rw descriptor and the use of the Aw descriptor for non-character data are available primarily for programs that were written before a true character type was available. Other uses include adding labels to binary files and the transfer of data whose type is not known in advance.

List items can be of type real, integer, complex, or logical. For character use, the data's binary form is converted to or from ASCII codes. Note that a variable's conventional numeric form (as used in mathematics, depending on its data type) is not used for ASCII interpretation.

w specifies a field width of 1 through 8 eight-bit characters, or 1 through 16 for complex values. Complex items use two storage units (Cray words) and require two A descriptors, for the first and second storage units respectively.

The Aw descriptor works with noncharacter data in essentially the same way as described in 8.7.1. The Rw descriptor works like Aw with the following 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.

The ANSI Fortran Standard does not provide for the use of A with noncharacter list items.

Examples:

In the following examples, characters shown here signify the corresponding ASCII codes used in storage or transmission; Ø is a binary 0, not a code for the 0 character; and ‿ represents a blank character.

Input Field Positions 1 2 3 4 5 6 7 8 9 10 12	Item Type	A or R Descr.	Internal Representation
A B C D E F G H I J K L	Integer	A8	ABCDEFGH
		R8	ABCDEFGH
A B C	Integer	A3	ABCØØØØØ
		R3	ØØØØØABC
A B C D E F G H I J K	Complex	A8,A3	ABCDEFGHIJKØØØØØ

Internal Representation	Item Type	A or R Descriptor	Output Field Positions 1 2 3 4 5 6 7 8 9
ABCDEFGH	Integer	A8	A B C D E F G H
		R8	A B C D E F G H
ABCDEFGH	Real	A9	‿ A B C D E F G H
		R9	‿ A B C D E F G H
ABCDEFGH	Integer	A6	A B C D E F
		R6	C D E F G H

Code examples:

The following examples are typical of older codes that were written before character type was available.

For the Aw descriptor:

```
      INTEGER IMAGE(10)
      READ (*,901) IMAGE
  901 FORMAT (10A8)             ! Accommodates an 80-column card
```

The above code would write a character string into memory as follows (where characters represent their ASCII codes in storage and Ø is a binary 0, not an ASCII code for the 0 character):

```
A B C D E F G H
I J K L M N O P
Q R S T Ø Ø Ø Ø
Ø Ø Ø Ø Ø Ø Ø Ø
...
```

For the Rw descriptor:

```
      INTEGER IMAGE(80)
      READ (5,902) IMAGE
  902 FORMAT (80R1)
      IF (IMAGE(1).EQ.65) GOTO...
```

The above code would write one character per word of memory, right-justified, as shown below (represented as in previous example):

```
Ø Ø Ø Ø Ø Ø Ø A
Ø Ø Ø Ø Ø Ø Ø B
Ø Ø Ø Ø Ø Ø Ø C
Ø Ø Ø Ø Ø Ø Ø D
...
```

With the character type, the above code would be written as follows:

```
      CHARACTER*80 IMAGE
      READ (5,902) IMAGE
  902 FORMAT(A80)
      IF (IMAGE(1:1).EQ.'A') GOTO ...
```

E.4 PUNCH STATEMENT

The PUNCH statement is a data transfer output statement. Under COS, the default output dataset is $PUNCH. The format is as follows.

```
PUNCH f [,iolist]
```

f Format identifier

iolist Input/output list specifying the data to be transferred

E.5 TYPE DECLARATION

Outmoded means of declaring data types are the DOUBLE statement for double precision, and the type statement including data length.

E.5.1 DOUBLE DECLARATION STATEMENTS

The form of the DOUBLE declaration type statement is:

```
DOUBLE v[,v]...
```

DOUBLE Desired data type

v Name of a constant, variable, array, function, or dummy procedure name; or an array declarator

The form of the double-precision FUNCTION statement is:

```
DOUBLE FUNCTION fun[([d[,d]...])
```

fun Symbolic name of the function subprogram in which the FUNCTION statement appears

d Dummy argument representing a variable, array, or external procedure name

E.5.2 TYPE STATEMENT DATA LENGTH

The formats of a type declaration with data length included are as follows.

```
type [*n] v[,v]...
```

IMPLICIT *type*[**n*](a_1[-a_2][,a_3[-a_4]]...)

  [,*type*[**n*](a_5[-a_6][,a_7[-a_n]]...)]...

type Can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[**len*] *len* is the length of a character entity and can be an unsigned, nonzero, positive integer constant expression.

**n* Data length as shown in table E-1. Any other data length gives a fatal error.

v Name of a constant, variable, array, function, or dummy procedure; or an array declarator.

a Letter or range of letters denoted by the first and last letters of the range separated by a hyphen. A range (a_1-a_n) has the same effect as a list of the letters (a_1,a_2,...a_n).

Table E-1. Data Length

Data Type \ n	1	2	4	8	16
INTEGER		64-bit†	64-bit integer		
REAL			64-bit real single precision††		128-bit real double prec.
COMPLEX			64-bit complex single precision		
LOGICAL		64-bit logical			
DOUBLE PRECISION					128-bit real double prec.

† This usage causes a warning message to be issued.
†† REAL*8 causes a warning message to be issued.

E.6 DATA STATEMENT FEATURES

The DATA statement has the following features, in addition to those described in 4.4:

- One constant must exist for each element of a whole array named in an *nlist,* unless the array is the last item in the list. In this case, the values in *clist* can specify any number of consecutive array element values, beginning with the first.

- If a variable, array element, or 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.

- An *nlist* entity of type logical can correspond to a *clist* constant of type Hollerith. A character constant can be specified to correspond to entities of any type except logical.

- A Hollerith constant (including the character literal form, '...') can initialize more than one element of an integer or real array if the array is specified without subscripts. For example, if an array is declared by INTEGER A(2), the following DATA statements have the same effect:

```
DATA A/'1234567890123456'/
DATA A/'12345678','90123456'/
```

E.7 IF STATEMENTS

Outmoded IF statements are the two-branch arithmetic IF and the indirect logical IF.

E.7.1 TWO-BRANCH ARITHMETIC IF

A two-branch arithmetic IF statement transfers control to statement s_1 if expression *e* is evaluated as nonzero or to statement s_2 if *e* is zero.

IF(*e*)s_1,s_2

e	Integer, real, or double-precision expression
s_1,s_2	Labels of executable statements in the same program unit

Example:

```
IF (I+J*K) 100,101
```

E.7.2 INDIRECT LOGICAL IF

An indirect logical IF statement transfers control to statement s_t if logical expression *le* is true and to statement s_f if *le* is false.

IF(*le*)s_t,s_f

le Logical expression

s_t,s_f Labels of executable statements in the same program unit

Example:

```
IF(X.GE.Y)148,9999
```

CREATING NON-FORTRAN PROCEDURES F

Functions and subroutines written in Cray Assembly Language (CAL), Cray C, and Cray Pascal can be used with CFT77 programs. This section lists the primary considerations in making external procedures compatible with CFT77.

F.1 CAL

Specific macros are available to aid the CAL programmer in writing routines to be used with CFT77. These macros maintain compatibility with different versions of Fortran. Following are some CAL linkage macros and their purposes:

- DEFARG defines argument transmission
- DEFB defines B register use
- DEFT defines T register use
- ENTRY defines CFT77-callable entry points
- ARGADD allows argument retrieval
- LOAD allows local variable reference
- STORE allows local variable updates

See the Macros and Opdefs Reference Manual, CRI publication SR-0012, for more information on linkage macros.

F.2 CRAY PASCAL

To use Pascal routines with CFT77, consider the following differences between Pascal and Fortran:

- Because all CFT77 arguments are passed by address, all arguments in Pascal must be declared VAR.

- Pascal and Fortran store arrays differently. Multidimensional arrays defined in one language are transposed in the other.

- Cray Pascal is a stack-based language. The Cray Pascal compiler uses the standard Cray stack management routines, so CFT77 routines also using the stack will not have a problem. A CFT77 program executing in static mode that calls a Pascal routine will have a stack allocated for the Pascal routine.

- If a Pascal routine performs any I/O with the default input or output files and the main routine is not a Pascal routine, the Pascal routine must reset (for input) or rewrite (for output) appropriately.

- A reference to a Pascal procedure name is a level of indirection more than a reference to a CFT77 subroutine name, when the name is specified as an argument. For example, the following Fortran and Pascal programs pass different information to B.

```
      SUBROUTINE A
      EXERNAL P
      CALL B(P)
      END
      SUBROUTINE B(P)
      EXTERNAL P
      CALL P
      END
```

```
program A;
    procedure B (procedure P);
    begin;
    end;
    procedure P; external;
begin
    B(P)
end;
```

  In CFT77 the parcel address of P is passed to subroutine B. In Pascal, a parameter descriptor containing the address of a word containing the parcel address of P is passed to procedure B.

F.3 CRAY C

To use C routines with CFT77, consider the following differences between C and Fortran:

* Because all CFT77 parameters are passed by address, all parameters in the C routines must be declared as "pointer to." This is true for all single-word items. Arrays and structures declared in C are correctly passed by address.

* CFT77 arrays and C arrays have different storage allocation. Multidimensional arrays defined in one language are transposed in the other.

* The C character pointer is incompatible with the CFT77 character type. Character values should not be passed between C and CFT77 routines.

* Cray C is a stack-based language. The Cray C compiler uses the standard Cray stack management routines, so CFT77 routines also using the stack will not have a problem. A CFT77 program executing in static mode that calls a C routine will have a stack allocated for the C routine.

* The CFT77 calling routine expects the C routine's name to be in uppercase. A lowercase C routine name produces an unsatisfied external reference.

See the Cray C Reference Manual, publication SR-2024, for more information about using C routines in CFT77 programs.

MACHINE REPRESENTATION OF DATA G

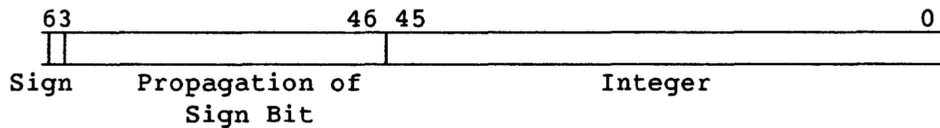
This section shows how words of storage are used to represent values of different data types. Numbers shown above the formats are bit positions, which represent powers of two in binary notation. Code that depends on internal representation is not portable and does not conform with the ANSI standard.

G.1 INTEGER TYPE

All integer data, whether 46 bits or 64 bits, is twos complement and is represented as illustrated in figure G-1.

46-bit integer

Range: $-2^{46} \leq I < 2^{46}$ or (approximately) $-10^{14} < I < 10^{14}$



64-bit integer

Range: $-2^{63} \leq I < 2^{63}$ or (approximately) $-10^{19} < I < 10^{19}$

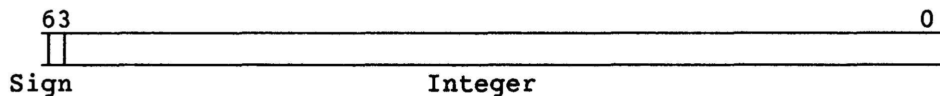


Figure G-1. Integer Data Formats

G.2 REAL TYPE

Real (floating-point) numbers are represented in a packed representation of a binary mantissa and an exponent (power of two). The mantissa is a 48-bit signed fraction. The sign of the mantissa is separated from the rest of the mantissa as shown in figure G-2. Since the mantissa is a signed magnitude, it is not complemented for negative values.

```
 63          48 47                              0
 |_|__________|_________________________________|
Mantissa  Exponent             Mantissa
  Sign          ↑
             Assumed
           Binary Point
```

Figure G-2. Floating-point Data Format

The exponent portion of the floating-point format is represented as a biased integer in bits 62 through 48. The bias that is added to the exponents is 40000_8. The positive range of exponents is 40000_8 through 57777_8. The negative range of exponents is 37777_8 through 20000_8. Thus, the unbiased range of exponents is the following (note the negative range is one larger):

2^{-20000_8} through 2^{+17777_8}

Example:

A real value of 10.0 is represented as follows.

Hexadecimal: 4004A00000000000
Octal: 0400045000000000000000

Binary:

```
0100000000000100101000000000000000000000000000000000000000000000
                ↑
                Bit 47
```

The leftmost bit, with a 0 value, indicates a positive mantissa; that is, the real value is positive. The next bit, set to 1, is in effect a sign bit for the exponent. It indicates a positive exponent value, meaning that the number's absolute value is 1.0 or greater. The value 4 in the exponent (100 appearing to the left of bit 47) means that the binary fraction in the mantissa is multiplied by 2^4 (Or, to express it another way, the binary point is moved 4 bits to the right from the mantissa's highest bit.) The first four bits of the mantissa, 1010, then indicate the real value of 10.0.

In terms of decimal values, the floating-point format of the CPU allows accurate representation of numbers to about 15 significant decimal digits in the approximate decimal range of 10^{-2466} through 10^{+2466}.

A zero value is not biased and is represented as a word of all zeros.

G.2.1 NORMALIZED FLOATING-POINT NUMBERS

A nonzero floating-point number is normalized if the most significant bit of the mantissa is nonzero. This condition implies the mantissa has been shifted as far left as possible and the exponent adjusted accordingly. Therefore, the floating-point number has no leading zeros in the mantissa. The exception is that a normalized floating-point zero is all zeros.

When your program creates a floating-point number by inserting an exponent of 40060_8 into a 48-bit integer word, you should normalize the result before using it in a floating-point operation. To do this, add the unnormalized floating-point operand to 0. However, CFT77 optimization suppresses an operation like X=X+0. ; you can perform it with code like the following:

```
DATA REALZERO/0./
X = X + REALZERO
```

G.3 DOUBLE-PRECISION TYPE

A double-precision value is represented by two words. The first has the same format as the real type, shown in figure G-3. The second word uses bits 0-47 as 48 additional bits of the mantissa. The other 16 bits of the second word must be zeros.

Range: $2^{-8193} \leq D \leq 2^{8190}$

Approximate decimal range: $10^{-2466} < D < 10^{2465}$

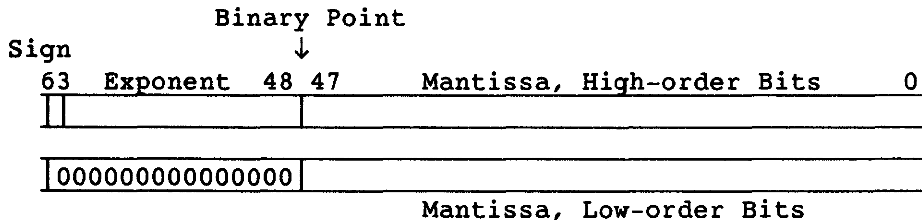


Figure G-3. Double-precision Format

G.4 COMPLEX TYPE

A complex value is represented by two words, each of which has the same format as the real type, shown in figure G-4. The first word represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

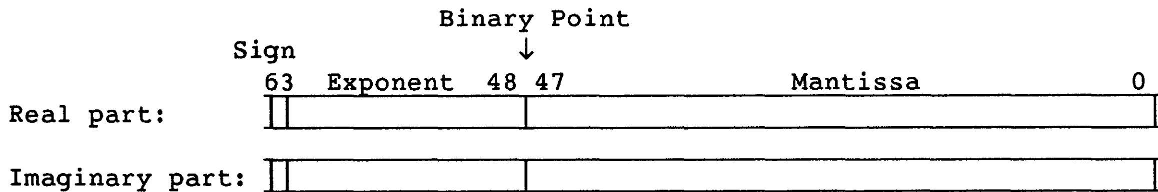


Figure G-4. Complex Format

G.5 CHARACTER TYPE

Characters are represented by 8-bit ASCII codes packed eight per word, as shown in appendix A.

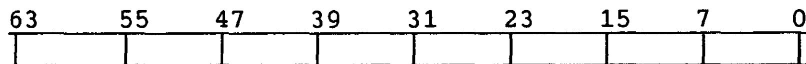


Figure G-5. Character Format

G.6 LOGICAL TYPE

A logical variable uses one 64-bit Cray word. Its value is represented differently under COS and UNICOS:

Computer System	True	False	Normal
CRAY-2	Nonzero	Zero	1 and 0
Not CRAY-2	Negative	Non-negative	-1 and 0

CRI does not guarantee a particular internal representation of logical values on any machine or system; CFT77 is designed on the assumption that logical values will be used only as described in the ANSI standard. Therefore it is not good programming practice to exploit gaps in type checking, such as between a function reference and its function value, to use logical values as numbers or vice versa.

DIFFERENCES BETWEEN CFT77 AND CFT H

This section lists differences between the CFT and CFT77 compiler that are likely to be of interest to users.

H.1 FUNCTIONAL DIFFERENCES

The following are differences between CFT77 and CFT in such areas as computation and optimization:

- CFT77 optimization is significantly different from CFT's. Consequently small numeric differences because of rounding may occur.
- CFT attempts to compensate for the inaccuracy inherent in division of real values by forcing strong rounding if the division occurs in an integer context. Although this may appear to give better results in some cases, the results are inconsistent depending on whether the compiler can detect the use of integers. Because of this possible inconsistency, CFT77 does not use this technique.
- CFT77 deletes statements that cannot be executed. A warning message is printed if a deleted statement has a label that is referenced in an ASSIGN statement.
- CFT77's optimizations include strength reduction of exponentiation operations. Since this can cause numeric differences, a warning message is printed during compilation.
- CFT77 does not allow assignments to a DO variable within the DO loop.
- CFT77 does not allow the traditional form of Hollerith (such as 3HSAM) to be used as the identifier in a STOP or PAUSE statement, although the quoted form of Hollerith ('SAM'H) is allowed. This restriction eliminates a possible ambiguity (such as STOP3H=1.).
- With CFT77, IMPLICIT NONE applies to implied-DO variables. With CFT, it does not. CFT77 does not support IMPLICIT SKOL.

- CFT77 and CFT treat pointers differently. Pointer is a separate data type in CFT77 rather than being integer as in CFT. Therefore operations with pointers are more restricted with CFT77. Pointers can be assigned to and from integers, but not to and from reals. Integers may be added or subtracted from pointers, and pointers may be added or subtracted, both with integer results. A program that first declares a variable as integer and then as a pointer causes an error with CFT77 but is correct with CFT.

- The default integer size for CFT77 is 46 bits; with CFT it is 64 bits. The INTEGER=64 or -i64 option on the control statement may be used to specify 64-bit precision. CFT's INT24 directive is not supported, but A-register arithmetic is used whenever possible.

- CFT77 generates explicit calls to the system heap manager for automatic arrays, array syntax temporaries, and some character code, even when ALLOC=STATIC is used. CFT never calls the heap manager directly.

- Character variables in CFT are limited to a length of 16383, and no more than 511 different lengths of character strings may appear in a single program unit. CFT77 does not have these restrictions.

- Three features proposed for the next Fortran standard have been included in CFT77 that are not in CFT:

  - Longer identifier names (up to 31 characters for internal names) which may contain underscores

  - A subset of the array syntax

  - Automatic arrays

- The two compilers have different compiler directives and control statement options. CFT directives that are not used by CFT77 are recognized, and warning messages are given that these are not implemented. The IVDEP directive must immediately precede a DO loop with CFT77, and directives cannot be continued. Bounds checking is disabled by the NOBOUNDS directive instead of BOUNDS(). The interaction between control statement options and compiler directives is also different with CFT77.

- When list-directed output is used, several copies of the same value may be printed with a repetition count. This occurs at different times with CFT and CFT77, although both of the compilers are in accord with the standard.

- CFT77 optimization precludes the use of implicit association. This is discussed in 4.3.5.

- CFT77 does not vectorize a loop containing more than one reference to the RANF function; CFT does vectorize such a loop. This can cause widely varying results. This is discussed at the end of appendix B.

H.2 SYNTAX AND ERROR DETECTION

Compile-time error detection is more extensive with CFT77 than with CFT. The following cases are errors with both compilers but are either not diagnosed by CFT or are handled differently by the two compilers.

1. CFT77 gives an error message for arguments to CLOCK, JDATE, and DATE that are not REAL or INTEGER; CFT does not.

2. CFT77 gives an error message when extraneous parentheses appear in PARAMETER statements: the form PARAMETER (*list*),(*list*),... is flagged as an error. CFT does not detect an error for this form.

3. CFT77 does not permit extra sets of parentheses around the input/output lists in I/O statements or the implied-DO list of array element names in DATA statements. CFT does not detect an error in these cases. The following examples have illegal extra parentheses:

```
WRITE(6,12) (A,B)      WRITE(6,12)((A(I),B(I)),I=1,10)
ENCODE(N,12,A)(B,C)    DATA ((A(I),B(I)),I=1,10)
```

4. CFT77 gives an error when an array is used as a statement function dummy argument; CFT does not detect this. CFT77 also does not allow an array name to be an actual argument to a statement function; an array element must be used.

5. CFT77 gives an error when a name that has been used as a function is later used as a subroutine (or vice versa); CFT does not.

6. CFT77 gives a warning message if it encounters the REC= or IOSTAT= specifiers in a NAMELIST I/O statement; CFT does not. Both compilers ignore the specifiers.

7. CFT77 gives an error if it detects a NAMELIST group name in the FMT= specifier of a READ or WRITE control item list; CFT does not.

8. CFT77 detects whether a statement label on an ELSE statement is referenced and gives an error message if it is. CFT produces a warning message if it encounters a statement label on an ELSE statement and then ignores the label.

9. CFT77 gives an error if there is no EXTERNAL statement for a dummy procedure that is passed as an actual argument. CFT does not detect an error if it sees a use of the name in a CALL statement or as a function reference before it sees the name used as an actual argument.

10. CFT77 enforces the requirement that a dummy argument used in an adjustable array bound must appear in every dummy argument list in the program unit that contains the array name; CFT does not enforce this.

11. CFT77 produces an error message when an implied-DO variable in a DATA statement is not an integer; CFT does not detect this error.

12. CFT77 prohibits an out-of-bounds array reference in an EQUIVALENCE statement and gives an error message. CFT gives a warning message unless the out-of-bound subscript expression is less than the lower bound for the dimension; CFT produces an error in this case.

13. CFT77 disallows an attempt to dimension a variable that has previously appeared (as a scalar) in a DATA statement. CFT does not detect this error.

14. CFT77 detects PARAMETER, DATA, or FORMAT statements appearing before a SUBROUTINE or FUNCTION statement in a program unit. CFT does not detect this.

15. CFT77 rejects the use of a logical third argument to the intrinsic functions CVMGP, CVMGM, CVMGZ, and CVMGN; CFT does not detect this error.

16. CFT77 detects illegal mixing of logical with other types in masking expressions or as arguments to logical intrinsics; CFT does not. This use is invalid with both compilers since the bit representation used for logical is not guaranteed and may change between releases or machines. In fact, the internal representation for logical is different on the CRAY-2 computer system than on the CRAY X-MP computer system.

17. CFT77 issues a CAUTION when an intrinsic function is explicitly typed and the declared type differs from the intrinsic's type. CFT issues a CAUTION whenever an intrinsic function is explicitly typed.

18. CFT77 does not allow assigning different lengths to variables in a type statement for noncharacter types. CFT does not detect this error. For example, the following statements produce errors with CFT77 but not with CFT:

    ```
    COMPLEX*8 X, Y*16

    REAL*8 X,Y
    ```

19. CFT77 gives an error message, but CFT does not detect, when a character array is dimensioned in a CHARACTER statement by the following non-standard construct:

    *name * length (dimensions)*

20. CFT77 does not allow an ENTRY name in a subroutine subprogram to appear in a type statement. CFT does not detect this error.

21. CFT77 allows only integer and Boolean expressions in a computed GOTO statement and gives an error if any other type is used. CFT does not detect the use of character or logical types and allows the use of other arithmetic types.

22. CFT77 rejects the use of an external name as an argument to the LOC function; CFT does not detect this error. Make LOC external if you need to do this.

23. CFT77 gives an error if END= is specified on a random access READ statement. CFT does not detect this error; the library routine ignores the specifier in this case.

24. Overindexing of arrays as a means of achieving dynamic common storage is allowed but causes a message to be issued. Heap storage is suggested for this purpose, as discussed in appendix D. The following generates a warning with CFT77 but not with CFT:

```
COMMON A, B, C(1500)
DIMENSION R(1)
EQUIVALENCE (R(1),C(1))
...
```

INDEX

INDEX

$ format descriptor (carriage control), 8-31
-a cft77 command parameter, 1-6
    stack, 2-22, 4-3, 4-26, 4-35, 4-43
    and ALLOC directive, 1-35
a.out UNICOS file, 1-2
a/A compiler option, 1-10 (UNICOS), 1-24 (COS)
ABORT and Flowtrace, 1-34
ABS, intrinsic function, B-2
Absolute value, intrinsic function, B-2
ACCESS command, for Cray-resident dataset, 1-14, 1-21
Access method, INQUIRE specifier, 7-34
ACCESS=, OPEN specifier, 7-30
ACCOUNT control statement, 1-14
ACOS, intrinsic function, B-6
ACQUIRE command, for remote file, 1-21
Actual argument, 2-33
    in function reference, 2-13
    in statement function reference, 2-17
    passed through subprogram, 2-32
Actual array, 4-5
    as pointee, 3-18
Address reference Table, 1-39
Adjustable array, 4-5, 4-11
AIMAG, intrinsic function, B-3
AINT, intrinsic function, B-2
ALLOC directive, 1-35
ALLOC parameter, CFT77 statement, 1-19
    STACK option, 2-22, 4-26, 4-3, 4-35, 4-43
ALOG and ALOG10, intrinsic function, B-7
Alphanumeric character set, 2-1
Alternate return, 2-31
    specifier, 10-4, 2-26 to 2-34
AMAX, intrinsic function, B-5, 2-37
American National Standards Institute (ANSI), 1-1
AMIN, intrinsic function, B-5, 2-37
AMOD, intrinsic function, B-2
AND, intrinsic function, B-8
ANINT, intrinsic function, B-2
ANSI (American National Standards Institute), 1-1
    messages about nonstandard usage, 1-10, 1-23
Apostrophe format descriptor, 8-27
Arguments, 2-32
    overview, 2-11
    actual, for external procedures, 2-33
    association, 2-32, 4-32
    character type, 3-14
    dummy procedures, 2-35
        EXTERNAL statement, 2-36
Arguments (continued)
        INTRINSIC statement, 2-36
    dummy, 2-34
    for intrinsic function, 2-20
    number, intrinsic function, B-10
Arithmetic expressions, 5-2, 5-8
    array section in, 4-18
    assignment statement, 5-3
    data type, 5-8, 5-10 (table)
    factors, 5-7
    operands, 5-6
    operators, 5-5
    precedence of operators, 5-5
    primaries, 5-6
    terms, 5-7
    type conversion, 5-12
Arithmetic functions (table), B-2
Arithmetic IF statement, 10-4, 6-6
    in vectorization, 10-6
Arithmetic operands, 5-6
Arithmetic operations, data types in (table), 5-10
Arithmetic operators in expressions (table), 5-5
Arithmetic value (definition), 3-1
Array expression, 4-20
    array operands in intrinsic functions, 4-23
    conformance of array operands, 4-21
    not an argument, 2-20, 2-33
    not in statement function, 2-17
    order of operations, 4-21
Array section (CFT77 extension), 4-16
    indexed section selectors, 4-17
    name, 4-17, 4-16
    vector-valued section selectors, 4-18
Array, 4-4
    actual, 4-5
    adjustable, 4-5, 4-11
    and implicit association, 4-33
    arguments, 2-34, 4-12
    assumed-size, 4-5
    automatic, 4-6
    bounds checking
    bounds directives (BOUNDS and NOBOUNDS), 1-34
    constant, 4-5
    declarators, 4-7, 3-3
        in POINTER statement, 3-17
        in type statement, 3-4
    DIMENSION statement, 4-6
    dimension, 3-3
    dimension, 4-4
    dummy array, 4-12

Array (continued)
dummy, 4-5
element, 4-9
as argument, 2-33
name, as argument, 2-33
substring, 4-4
in EQUIVALENCE statement, 4-4, 4-34
equivalencing, 3-19
expression, 4-20
implied-DO list, 4-26
kinds (table), 4-4
names, use of, 4-13
pointee, 4-5
scope, 2-7
section, 4-16
size, 4-12
specification and size (figure), 4-14
storage sequence, 4-10
subscripts, 4-9
in type statement, 3-4
asa routine, 7-26
ASIN, intrinsic function, B-6
ASSIGN command, for COS datasets, 7-32
ASSIGN statement, 6-13
for FORMAT label, 7-18
Assigned GOTO statement, 6-12
Assignment, 5-1
arithmetic, 5-3
and array sections, 4-16
character, 5-13
data types (table), 5-4
logical, 5-19
relational, 5-16
in vectorizing, 10-4
Associated, term defined, 4-28
Association, 4-28, 4-32
of arguments, 2-32
overview, 2-11
via pointers, 3-19
Assumed-size array, 4-5
Asterisk
default unit, 7-17
format descriptor, 7-18
in CHARACTER statement, 3-11
ASYNCDR, 9-5
ASYNCMS, 9-5
ATAN and ATAN2, intrinsic function, B-6
Automatic array, 1-1, 1-6, 4-6
and static allocation, 1-6
element, in DATA statement, 4-28

-b parameter, cft77 command, 1-4, 1-6
B compiler option (UNICOS), 1-10
B= parameter, CFT77 statement, 1-19
BACKSPACE statement, 7-38
file identifier, 7-13
Backward format descriptor (TL), 8-28
Basic real constant, definition, 3-7
Binary output
UNICOS, 1-2, 1-4, 1-10
COS, 1-14, 1-17, 1-19
disabling, 1-10, 1-19
BL/NOBL directives, 1-32
Blank as zero, format descriptor (BZ), 8-30
Blank character, 4-1
control specifier, INQUIRE, 7-35
Blank common block, 4-38
and local common, 4-44
and pointers, 3-19
size, 4-38
BLANK= OPEN specifier, 7-30
BLANK= INQUIRE specifier, 7-35
BLCKDATA, name for block data subprogram, 4-45
$BLD dataset, 1-14, 1-19
BLOCK DATA statement, 4-45, 2-6, 2-11, 2-21
Block data subprogram, 2-21, 2-8, 4-44
name in EXTERNAL statements, 2-36
Block IF statement, 6-2, 6-6
in vectorization, 10-6
Blocked files and datasets, 7-12
Boolean type, 3-15
function, B-8 (table)
item, in expression, 5-10
operand, in masking expression, 5-23
versus logical type, 3-15
Bottom-loading, 1-32
Bounds checking, array, 1-11, 1-24
directives (BOUNDS and NOBOUNDS), 1-34
Branch backward, in vectorizing, 10-4
Branch, 10-4
BUFFER IN and BUFFER OUT statements, 9-1, 7-8, 7-10, 7-12
LENGTH function, 9-4
UNIT function, 9-3
with SETPOS, 9-7
BZ format descriptor (blank as zero), 8-30

-C parameter, cft77 command, 1-6, 1-25
C language, procedures in, F-2
C parameter, CFT77 statement (COS), 1-19
CABS, intrinsic function, B-2
CALL statement, 2-26, 2-8, 2-25
and alternate return, 2-31
dummy argument in, 2-28
timing with Flowtrace, 1-34
transfer back to, 2-26
vectorizing, 10-4
CAL
output, 1-9, 1-4 (UNICOS), 1-19, 1-17 (COS)
procedures in, F-1
vectorizable function in, 1-29
Cancellation, effect on accuracy, 3-7
Carriage control edit descriptor ($), 8-31
CCOS, intrinsic function, B-6
CDIR$, 1-27
also see Compiler directives
listing, 1-5, 1-18
CEXP, intrinsic function, B-7
CFT77 compiler, 1-1
conventions, 1-1
cross reference listings, 1-38
directives, 1-27

CFT77 compiler (continued)
Flowtrace, 1-33
language elements, 2-1
listable output, 1-37
using with COS, 1-13
using with UNICOS, 1-2
CFT77/CFT differences, I-1
functional differences, I-1
syntax, error detection, I-3
CHAR, intrinsic function, 2-37, B-4
Character set, 2-1, A-1
Character type, 3-10
arguments, 3-14, 3-10
array, as format specifier, 7-19
assignment statement, 5-14
association of entities, 4-33
CHARACTER statement, 3-11
asterisk specification, 3-11
in common block, 4-38
constant in DATA statement, 4-27
constant in PARAMETER statement, 4-2
delimiter, 3-10
entry name, 2-29
equivalence, 4-37
expression, 5-13
as actual argument, 2-20, 2-33
as format specifier, 7-18
evaluation, 5-15
in statement function, 2-17
format (figure), G-4
function, 3-10, 3-11, B-5 (table)
declarations, 3-12
length, 2-24
Hollerith, 5-15, 5-10, 8-27, E-2
internal file, 7-15
length, 3-5
machine representation, G-4
position in string, 3-10
primary, 5-13
and records, 7-10
set, A-1
storage unit, 4-28
substrings, 3-13
variable, 3-10
cigs, 1-7
CIGS, 1-20
cilist, 7-20
Circular shift, intrinsic function, B-9
Class, of symbolic name, 2-7
CLOCK, intrinsic function, B-10
CLOG, intrinsic function, B-7
CLOSE statement, 7-32
and EOD record, 7-10
file identifier, 7-13
specifiers (table), 7-31
CMPLX, intrinsic function, B-4, 2-37
CODE and NOCODE directives, 1-33
Code, see Source code or Generated code
Collating sequence, 2-1
Colon format descriptor, 8-29
Comment, source, 2-3
Comment lines, 2-11
Common block, 4-37
overview, 2-11
allocation, 1-19
Common block (continued)
association of arguments, 2-33
BLOCK DATA statement, 4-45
block data subprogram, 4-44
COMMON statement, 4-39
and DATA statement, 4-25
directive (DYNAMIC), 1-36
EQUIVALENCE statement, 4-37
LOCAL COMMON statement, 4-44
names, 4-38, 4-1, 4-3, 4-39, 3-3, 1-39
and statement function argument, 2-17
same as function name, 2-24
redefined in subprogram, 1-30
referencing, 4-39
SAVE statement, 4-32
size, 4-42
storage sequence, 4-40, 4-28
TASK COMMON statement, 4-43
Common logarithm, intrinsic function, B-7
COMMON statement, 4-39
association, 4-33
block data subprogram, 4-45
entities defined in subroutines, 2-26
pointer in list, 3-19
static variables, 4-3
storage, 4-26
COMMONS, SEGLDR directive, 7-40
Compiler options, 1-10 (UNICOS), 1-24 (COS)
enabling, 1-7 (UNICOS), 1-20 (COS)
local control, 1-33
Compiler directives, 1-27
array bounds checking, 1-34
dynamic common block, 1-36
flowtrace directives, 1-33
issues, 1-4 (UNICOS), 1-17 (COS)
listable output control, 1-32
range, 1-17
suppressing optimization, 1-31
vectorization control, 1-28
Compiling under UNICOS, 1-2
cft77 command, 1-5
compiler options, 1-10
Compiling under COS, 1-13
CFT77 control statement, 1-17
compiler options, 1-23
COMPL, intrinsic function, B-8
Complement, logical, intrinsic function, B-8
Complex type, 3-9
conjugate or imaginary portion, function, B-3
conversion, intrinsic function, B-4
conversion to and from, 5-12
format descriptor, 8-22
internal format (figure), G-4
machine representation, G-4
storage, 4-26
type statement, 3-4
Compress-index hardware, 10-5, 1-7, 1-20
Computed GOTO statement, 6-12
Concatenation
in actual argument, 3-14
operator // , 5-13
Conditional blocks, 6-1, 6-6
block IF statement, 6-4
ELSE statement, 6-5
ELSEIF statement, 6-4
ENDIF statement, 6-4

Conditional scalar merge, intrinsic
  function, B-9
Conditional vector merge functions, B-11
Conditional vector merge hardware, 10-5
Conformance of array operands, 4-21
    checking, 1-11 (UNICOS), 1-24 (COS)
CONJG, intrinsic function, B-3
Conjugate of complex value, intrinsic
  function, B-3
Connection specifier, INQUIRE, 7-34
Constant, character, 3-11
Constant array, 4-5
Constant expression, in declarators, 4-5
Constant expression, in PARAMETER
  statement, 4-2
Constant increment variable, 1-8, 1-22, 10-4
Constant-size array, 4-5
Constant, 2-2, 4-1
    in type statement, 3-4
    reference, 2-7
    commonly used, C-1
Continuation line, 2-3
CONTINUE statement, 6-9
Control information list, I/O, 7-20
Control statement under COS, 1-17
Conventions, 1-1
COS (cosine) intrinsic function, B-6
COS (Cray operating system)
    compiling under, 1-13
    dataset, 7-14; also see Dataset
    debugging under, 1-16
    listing file, 1-16
    range of units, 7-16
    object file, 1-10
COSH, intrinsic function, B-7
COT, intrinsic function, B-6
CPU parameter, CFT77 statement, 1-19
CPU targeting, 1-6 (UNICOS), 1-19 (COS)
CPU= parameter, CFT77 statement, 1-25
Cray Assembly Language, see CAL
CRAY X-MP, CPU targeting, 1-7 (UNICOS),
  1-19 (COS), 1-26
CRAY-2 systems
    memory paging, 1-10, 1-12
    and local common, 4-44
Cross reference listings, 1-38
    enabling, 1-12, 1-25
    labels, 1-41
    Parameter Table, 1-41
    source program references, 1-41
    symbol table, 1-38
Cross-compiling, 1-25
Cross-reference listings, 1-2
CRSUBMIT command, 1-13
CSIN, intrinsic function, B-6
CSMG, intrinsic function, B-9
CSQRT, intrinsic function, B-7
CVMGx, intrinsic functions, 2-37
CVMGx, intrinsic functions, B-11

-d cft77 command parameter, 1-7, 1-10
D compiler option, 1-10
D format descriptor (double precision),
  8-13, 8-20
    automatic disabling, 1-12
    overriding, 1-24
DABS, intrinsic function, B-2
DACOS, intrinsic function, B-6
DASIN, intrinsic function, B-6
Data, machine representation, G-1
Data Length (table), E-13
DATA statement, 4-23, 2-6
    allowed entities, 4-28
    CAL output, 1-9, 1-19
    data types in, 4-27
    and definition, 2-21, 4-30, 4-31
    with dummy argument, 2-34
    implied-DO list, 4-26
    in block data subprogram, 4-45
    memory allocation, 1-6, 1-19, 4-26
    outmoded features, E-13
    and static variables, 4-3
    task common variables, 4-43
Data structures and storage, 4-1
    arrays, 4-4
    association, 4-28
    common blocks, 4-37
    constants, 4-1
    DATA statement, 4-23
    storage, 4-28
    variables, 4-3
Data transfer statements (READ, WRITE,
  PRINT), 7-19
    control information list, 7-20
    end-of-file condition, 7-27
    error condition, 7-26, 7-27
    I/O list, 7-22
    implied DO list, 7-23
    operation, 7-24
    output to a printer, 7-26
    restrictions, 7-27
Data type, 3-1
    arithmetic expressions, 5-8
    arithmetic operations (table), 5-10
    array element, 3-3
    in assignment statements (table), 5-4
    Boolean, 3-15
    character, 3-10
        also see Character type
        format descriptor (A), 8-26
    complex, 3-9
        format descriptor, 8-22
    constant, 4-1
    conversion, 5-12
        to pointers, 3-17
        in PARAMETER statement, 4-2
        functions (table), B-4
    double-precision, 3-8
        format descriptor (D), 8-13, 8-20
    example values, 3-2
    exponentiation (table), 5-11
    functions, 2-14
    implicit, 3-3
    integer, 3-6
        also see Integer type
        64-bit, 1-7, 1-21, 1-35
        format descriptor (I), 8-21

Data type (continued)
logical, 3-9
format descriptor (L), 8-22
pointer, 3-16
real, 3-7
formatting, 8-13
input formatting (F, E, G), 8-17
output, 8-14 (F), 8-15 (E), 8-16 (G)
relational operations (table), 5-18
specification, 3-3
data length in statement, E-12
DOUBLE statement, E-11
function value, 2-23
IMPLICIT NONE statement, 3-6
IMPLICIT statement, 3-4
in a FUNCTION statement, 3-5
in block data subprogram, 4-45
pointee array, 3-17
statement function, 2-19
statements, 3-3, 3-5
DATAN, intrinsic function, B-6
Dataset, COS, 7-14
with BUFFER IN/OUT, 9-2
job input, 7-15
creating, 7-2
I/O, 7-11
name length, 1-17
fetching from front end, 1-21
local, 1-14
making local to job, 1-21
nondefault, 1-14
structure, 7-12
DATE, intrinsic function, B-10
Date functions (table), B-10
Date of compilation, 1-37
DBLE, intrinsic function, B-4, 2-37
DCOS, intrinsic function, B-6
DCOSH, intrinsic function, B-7
DCOT, intrinsic function, B-6
DDIM, intrinsic function, B-3
DEBUG keyword, CFT77 statement, 1-16, 1-22
DEBUG control statement, 1-16
debug utility (UNICOS), 1-4
Debugging, 1-4 (UNICOS), 1-16 (COS)
symbol table for, 1-10, 1-23
DECODE statement, E-6
Default
files (UNICOS) 1-3 (figure)
units, I/O, 7-17
values, cft77 command, 1-5
values, CFT77 control statement, 1-18
Definition, 4-30
entities in subprograms, 2-21
DELETE, CLOSE status, 7-31, 7-32
Delimiter, character, 3-10
Dependencies, vector, 1-28
DEXP, intrinsic function, B-7
Difference, positive, intrinsic function, B-3
Differences between CFT77 and CFT, I-1
DIM, intrinsic function, B-3
Dimension declarator and bound expression, 4-7
DIMENSION statement, 4-6
and DATA statement, 4-23
in block data subprogram, 4-45
DINT, intrinsic function, B-2
DIRECT, file status, 7-30
Direct file access, 7-36; also see File access
Direct-access I/O, record identifier, 7-21
DIRECT=, INQUIRE specifier, 7-35
Directives
control of command options, 1-33
issues, 1-4 (UNICOS), 1-17 (COS)
listing, 1-18
range, 1-17
scalar optimization, 1-31
vectorization, 1-28
DISPOSE control statement, 1-15
dispose/DISPOSE command, 7-2, 7-28, 1-21 (COS)
Division, prohibition on pointers, 5-10
DLOG, intrinsic function, B-7
DMAX1, intrinsic function, B-5, 2-37
DMIN1, intrinsic function, B-5, 2-37
DMOD, intrinsic function, B-2
DNINT, intrinsic function, B-2
DO loop, 6-6
array syntax, 1-1
DO statement, 6-8
ENTRY statement in, 2-29
loop control, 6-10
low trip count, 1-30
option j/J to force execution, 1-24
option to force execution, 1-11
prefetching operands, 1-32
terminal statement, 6-9
vectorizing, 10-4
directives, 1-28
DO variable, 6-6, 6-8, 6-10
in implied-DO list (I/O), 7-23
Dollar sign format descriptor (carriage control), 8-31
Double-precision type, 3-8
conversion to and from, 5-12
disabling, 1-12
exponent, 3-8, 5-11
format descriptor (D), 8-13, 8-20
machine representation (figure), G-3
product, intrinsic function, B-3
range, 3-8
type statement, 3-4
outmoded, E-11
value, storage, 4-29
DPROD, intrinsic function, B-3
DSIN and DSINH intrinsic functions, B-6
DSQRT, intrinsic function, B-7
DTAN and DTANH, intrinsic functions, B-6
Dummy argument, 2-35
as actual argument, 2-34
as alternate return, 2-31
character, 3-11
cross reference listing, 1-40
DATA statement, 4-28
definition, 2-32

Dummy argument (continued)
ENTRY statement, 2-29
function subprogram, 2-24
and pointers, 3-17
SAVE statement, 4-31
statement function, 2-17
subroutine, 2-26
Dummy array, 4-5, 4-12
adjustable, 4-13
Dummy procedure, 2-35
name, 2-7
in type statement, 3-4
subroutine name, 2-27
DUMPJOB statement, 1-16
$DUMP dataset, 1-16
Dynamic common block directive (DYNAMIC), 1-36
Dynamic allocation, and pointers, 3-16

-e cft77 command parameter, 1-7, 1-10
E format descriptor (real)
output, 8-13, 8-15
input, 8-17
E parameter, CFT77 statement, 1-20
Edit descriptors, see Format descriptors
EDN parameter, CFT77 statement, 1-21
EJECT directive, 1-33
ELSE block, 6-2
ELSEIF block, 6-2
ema/EMA (extended memory addressing)
optimization option, 1-3, 1-7 (UNICOS), 1-20 (COS)
cross-compiling for, 1-26
Embedded comment, 2-3
ENCODE statement, E-6
END statement, 6-15, 2-6, 2-11, 2-21, 2-22, 2-23, 2-26
and RETURN statement, 2-30
block data subprogram, 4-45
End-of-data (EOD) record, 7-10
namelist processing, 9-14
End-of-file, 7-27, 7-10, 7-38
continuing execution, 7-21
identifier, I/O, 7-21
namelist processing, 9-14
with implied-DO, 7-24
END= specifier, I/O statement, 7-21
ENDFILE statement, 7-10, 7-38
file identifier, 7-13
ENDIF statement, 6-2, 6-6
ENTRY statement, 2-29, 2-6, 2-21, 2-23, 2-26
association, 4-33
dummy argument in, 2-34
EOF, see End-of-file
.EQ. operator, 5-15, 5-18
Equivalence, logical, intrinsic function, B-8
EQUIVALENCE statement, 4-34, 4-40
and local common, 4-44
array names in, 4-36, 4-4
common storage, 4-42
and dummy argument, 2-34
restrictions, 4-36

EQV, intrinsic function, B-8
ERR= specifier, I/O statement, 7-21
CLOSE specifier, 7-31
INQUIRE specifier, 7-34
OPEN specifier, 7-30
Error, I/O, 7-20
EOF, 7-27
recovery, 7-27
with implied-DO, 7-24
Exclamation point (embedded comment), 2-3
Executable statement, 2-4
EXIST=, INQUIRE specifier, 7-34
EXIT control statement, 1-16
and Flowtrace, 1-34
EXP, intrinsic function, B-7
Exponent, real, 3-7
Exponentiation
data types in (table), 5-11
expression in PARAMETER statement, 4-2
functions (table), B-7
Expression, 5-1
as argument, 2-33, 3-14
character, 5-13
as actual argument, 2-20
in statement function, 2-17
logical, 5-17
masking, 5-22
operators (table), 5-5
relational, 5-15
vectorizable, and functions, 1-29
Extended memory addressing (EMA)
UNICOS, 1-3, 1-7
COS, 1-16, 1-20
cross-compiling, 1-26
EXTERNAL declaration, 3-6
External file, see File
External function, 2-22, 2-7, 2-8, 2-13
restrictions, 2-22
External procedure, 2-8
EXTERNAL statement, 2-36, 2-13, 2-19, 2-22, 2-35
entry name in, 2-29

F format descriptor (real), 8-13, 8-14, 8-17
f/F compiler option, 1-11 (UNICOS), 1-24 (COS), 1-33, 1-34
and PROGRAM statement, 2-12
Factors, in arithmetic expressions, 5-7
FETCH command for remote file, 1-21
FETCH control statement, 1-14, 1-15
file.f output file, 1-9
FILE=, OPEN specifier, 7-13, 7-30
File, 7-11
creating, 7-1, 7-2
default, 7-17, 1-3 (UNICOS), 1-14, 1-15 (COS)
direct access, 7-36
identifier, 7-13
in INQUIRE statement, 7-33
in OPEN statement, 7-30
as unit identifier, 7-16, 7-20
internal, 7-15

File (continued)
sequential access, 7-38
BACKSPACE statement, 7-39
ENDFILE statement, 7-39
REWIND statement, 7-40
structure, 7-12
FLOAT, intrinsic function, 2-37, B-4
Floating-point data format (figure), G-2
flodump command (CRAY-2 Flowtrace), 1-34
flow command, 1-34
FLOWMARK subroutine, 1-34
Flowtrace, 1-33, 10-1, 10-2
directives, 1-33
generating, 1-11, 1-24
FMT= specifier, 7-21
FORM=, INQUIRE specifier, 7-35
FORM=, OPEN specifier, 7-30
Format descriptor, 8-10, 8-12 (table)
A (character), 8-26
apostrophe, 8-27
asterisk, E-7
BN (ignore blanks), 8-30
BZ (blanks as zeros), 8-30
colon, 8-29
complex, 8-22
D (double precision), 8-13, 8-20
dollar sign (carriage control), 8-31
E (real), 8-15, 8-13, 8-17
F (real), 8-14, 8-13, 8-17
G (real), 8-16, 8-13, 8-17
H (Hollerith), 8-27
I/O lists and formats, 8-7
integer (I), 8-21
L (logical), 8-22
O (octal), 8-24
outmoded, E-7
P (scale factor), 8-20
positional, 8-28
quotation mark, 8-27
R (right-justified), E-8
real numbers, 8-13
S, SS (suppress + sign), 8-30
scale factor, 8-20
slash, 8-29
SP (include + sign), 8-30
T (character position), 8-28
TL (backward), 8-28
TR (forward), 8-28
X (forward), 8-28
FORMATTED, file status, 7-30
Formatted data assignment, E-5
also see Internal files, 7-15
DECODE statement, E-6
ENCODE statement, E-6
Formatted I/O, 8-5
descriptors, 8-9
format identifier, 7-21
FORMAT statement, 2-6, 8-6
label as specifier, 7-18
format, 7-18
also see Format descriptors
changing max length, 7-40
in INQUIRE, 7-35
tutorial, 7-5
Formatted record, 7-10
FORMATTED=, INQUIRE specifier, 7-35
Fortran language elements, 2-1
Also see Executable program
character set, 2-1
lines, 2-2
order of statements and lines, 2-5
Fortran language elements (continued)
statements, 2-4
symbolic names, 2-7
syntactic items, 2-2
Forward format descriptor (TR and X), 8-28
Front-end computer, transferring file to, 7-2
FTREF, 4-38
full, optimization option, 1-8
FULL, optimization option (COS), 1-22
FUNCTION statement, 2-24, 2-6, 2-11, 2-22, 2-23, 3-5
and character name, 3-11
argument used in statement function, 2-18
Function, 2-13, 2-8
array operands in, 4-23
and array sections, 4-16
Boolean, 3-15
character, 3-11
data type, 2-14
declarations, character, 3-12
execution, 2-15
external, 2-22
vector version, 1-29
generic, 2-19
name as argument, 2-20
in logical IF statement, 2-16
intrinsic functions, 2-19
referencing, 2-20
restrictions, 2-20
name, 2-7
as actual argument, 2-34
in ENTRY statement, 2-30
in type statement, 3-4
order of evaluation, 2-16
and pointees, 3-17
reference, 2-13
statement function, 2-16
subprograms, 2-22, 2-8, 2-13
and association, 4-32
calling itself, 2-17
character, 3-14
name, in DATA statement, 4-28
type, 3-5
value, 2-8, 2-23
definition, 4-31
.f extension on file names, 1-2

G format descriptor (real)
Output, 8-13, 8-16
Input, 8-17
g/G compiler option
UNICOS, 1-3, 1-5, 1-11
COS, 1-16, 1-18, 1-24

Gather-scatter hardware, 1-7 (UNICOS),
1-20 (COS), 10-5
.GE. operator, 5-15, 5-18
Generic function names, 2-20
in INTRINSIC statement, 2-37
GETPOS, 9-5
GETWA, 9-5
Global data, 4-34
Global name, 2-7
Global variable (as in Pascal), 4-38
GO parameter, SEGLDR statement, 1-14
GOTO statements, 6-11
assigned, 6-12
computed, 6-12
in vectorization, 10-6
unconditional, 6-11
.GT. operator, 5-15, 5-18

H format descriptor (Hollerith), 8-27
h/H compiler option
UNICOS, 1-3, 1-5, 1-11
COS, 1-16, 1-18, 1-24
and PROGRAM statement, 2-12
Header statement, 2-21, 2-4
dummy argument in, 2-34
Heap allocation, 1-6, 1-19, 1-30, H-1
Hollerith type, 5-15, E-2
expression, E-4
relational expresssion, E-4
format descriptor (H), 8-27
in file id, 7-13
in expression, 5-10
in masking expression, 3-16, 5-22
in relational expression, 5-18
Hyperbolic trig, intrinsic function, B-7

-i parameter, cft77 command, 1-7, 3-6
and INTEGER directive, 1-35
I option, cft77 command, 1-10
I parameter, CFT77 statement, 1-21
I/O 7-1 (tutorial), 8-1, 9-1
BUFFER IN/OUT statements, 9-1
buffer lengths, changing, 7-40
CLOSE statement, 7-32
datasets, 7-11
default units, 7-17
extensions, 9-1
files, 7-11
access methods, 7-36
internal, 7-15
structures, 7-12
format descriptors, 8-10
formats, 7-18
formatted I/O, 8-5
format descriptors, 8-10
INQUIRE statement, 7-33
internal files and records, 7-15
list-directed I/O, 8-2
list, 7-22
changing max length, 7-40
NAMELIST statement, 9-10
OPEN statement, 7-28

I/O (continued)
performance, 7-8
comparison, random I/O, 9-6
PRINT statement, 7-19
random, 9-5
READ statement, 7-19
records, 7-10
statements (table), 7-9
assumed-size array in, 4-6
unformatted I/O, 8-1
units, 7-16
WRITE statement, 7-19
IABS, intrinsic function, B-2
ICHAR, intrinsic function, B-4, 2-37
IDIM, intrinsic function, B-3
IDINT, intrinsic function, B-4, 2-37
IDNINT, intrinsic function, B-2
IF statements
arithmetic, 6-6
two-branch, E-14
block, 6-4, 6-2
functions in, 2-16
IF level, 6-2
logical, 6-5
indirect, E-14
as loop terminal statement, 6-8
relational expression in, 5-15
structure (figure), 6-2, 6-3
IFIX, intrinsic function, 2-37, B-4
Imaginary part (complex) range, 3-9
Implicit association, 4-33
IMPLICIT statement, 2-6, 2-14, 3-4, 4-2
in block data subprogram, 4-45
IMPLICIT NONE, 3-6
Implied DO list
DATA statement, 4-26
I/O, 7-23
and I/O speed, 7-8
INCLUDE statement, 1-36
Increment value, 6-6
Incrementation count, also see Trip count,
6-6
INDEF parameter, CFT77 statement, 1-23
INDEX, intrinsic function, B-5
Initial line, 2-3
Initial values, in DATA statement, 4-23
Initially defined variable, 4-31
Input/Output, see I/O
INQUIRE statement, 7-33
specifiers (table), 7-34
INT, intrinsic function, B-4, 2-37
INTEGER directive, 1-35
INTEGER= parameter, CFT77 statement, 1-21,
3-6
Integer type, 3-6
constant expression, 5-2
constant, 3-6
conversion to and from, 5-12
intrinsic function, B-4
to pointers, 3-17
data formats (figure), G-1
expressions and pointers, 3-17
format descriptor, 8-21
INTEGER directive, 1-35

Integer type (continued)
machine representation, G-1
nearest, intrinsic function, B-2
operation, 5-10
quotient, 5-5
range, 3-6
selecting 64-bit, 1-7 (UNICOS), 1-21 (COS), 1-35
storage, 4-28
type statement, 3-4
Internal files, 7-15
Intrinsic function, 2-19, 2-13, 2-8, B-1
array operands in, 4-23
as dummy arguments, 2-35
Intrinsic function (continued)
referencing, 2-20
replacing with same name, 2-36
restrictions, 2-20
INTRINSIC statement, 2-36
and dummy argument, 2-35
Invariant value or expression, in vectorizing, 10-4
$IN dataset, 1-15, 1-16, 1-21, 7-15, 7-17
$IOLIB, errors detected by, 7-27
IOSTAT=, specifier, I/O statement, 7-21, 7-30, 7-31, 7-34
IRTC, intrinsic function, B-10
ISIGN, intrinsic function, B-2
IVDEP directive, 1-28

j/J compiler option, 1-11 (UNICOS), 1-24 (COS), 6-6
JCL file, 1-13
JDATE, intrinsic function, B-10
JN parameter, JOB statement, 1-13
JOB control statement, 1-13
Julian date, intrinsic function, B-10

KEEP, CLOSE status, 7-31
Keyword, 2-2

-l parameter (listing file), cft77 command, 1-7
L format descriptor (logical), 8-22
L/LIST compiler option
UNICOS, 1-10, 1-3, 1-5
COS, 1-21, 1-16, 1-18
Label Cross-reference Table, 1-41
Labels, 2-2
LDR control statement, common block size, 4-42
.LE. operator, 5-15, 5-18
Leading zeros, intrinsic function, B-9
LEADZ, intrinsic function, B-9
LEN, intrinsic function, B-5
Length, character value, 3-10, 3-11
LENGTH function, 9-4, 9-1
Lexical relation, intrinsic function, B-5
LGE, intrinsic function, B-5, 2-37
LGT, intrinsic function, B-5, 2-37
LIBIO, errors detected by, 7-27
Limit, of DO variable, 6-6
Line number in source, 1-41
Lines, 2-2
comment, 2-3
compiler directive, 2-4
continuation, 2-3
initial and terminal, 2-3
Linked lists, 3-16
LIST and NOLIST directives, 1-33, 1-18
LIST keyword, CFT77 statement, 1-21, 1-37
List-directed I/O, 8-2
tutorial, 7-4
Listable output, 1-37
directives (EJECT, LIST, CODE), 1-33
file, 1-7 (UNICOS), 1-21 (COS)
INCLUDE file, 1-37
options, 1-10 (UNICOS), 1-23 (COS), 1-3, 1-4, 1-16
conflicting, 1-5, 1-18
Literal constant, 4-1
LLE, intrinsic function, B-5, 2-37
LLT, intrinsic function, B-5, 2-37
Loader, and common block size, 4-42
Also see SEGLDR
LOC function, 3-18, 3-19, B-10
Local and global data, 1-4
LOCAL COMMON statement, 4-44, 4-38
Local data, 4-34, 1-17
Local dataset, 1-14
Local name, 2-7
Local memory paging (CRAY-2), 1-10, 1-12
Location, intrinsic function, B-10
LOG, intrinsic function, B-7, 2-37
LOG10, intrinsic function, B-7, 2-37
Logarithmic functions (table), B-7
Logical expression, 5-17
array section in, 4-21
assignment statement, 5-19
components of, 5-21
operations (table), 5-24
operators, 5-19, 5-20 (table)
as functions (table), B-8
for masking expressions, 5-22
Logical IF statement, 6-5
in vectorization, 10-5
storage, 4-28
Logical shift, intrinsic function, B-9
Logical type, 3-9
entity, in DATA statement, 4-27
format descriptor (L), 8-22
machine representation, G-4
type statement, 3-4
versus Boolean type, 3-15
Loopmark, 1-12 (UNICOS), 1-24 (COS), 1-2, 1-15
Loop, DO, 6-6; also see DO loop
.LT. operator, 5-15, 5-18

-m cft77 command parameter, 1-8
m/M compiler option
UNICOS, 1-2, 1-3, 1-12, 1-37
COS, 1-18, 1-24, 1-37

Machine representation, G-1
    character type, G-4
    complex type, G-4
    double-precision type, G-3
    integer type, G-1
    logical type, G-4
    real type, G-2
Main program, 2-8
    scope of name, 2-7
MASK, intrinsic function, B-8
Masking expressions, 5-22, 5-24 (table), 3-15
    with Hollerith operand, 3-16
MAX, intrinsic function, 2-37, B-5
Maximum/minimum functions (table), B-5
Memory management, D-1
Memory, see Storage
Messages
    Levels, 1-8 (UNICOS), 1-20 (COS)
    COS dataset, 1-21
    listings, 1-37
MIN, intrinsic function, B-5, 2-37
MOD, intrinsic function, B-2
Mode, BUFFER IN/OUT statements, 9-2
Multiplication, 5-5
    prohibition on pointers, 5-10
    rounding, 1-12
Multitasking, 10-3
    and storage, 4-29

NAME=, INQUIRE specifier, 7-34
NAMED=, INQUIRE specifier, 7-34
NAMELIST statement, 9-10
    and entry name, 2-29
    input, 9-12
    name, 2-6
    output, 9-17
    processing, 9-14
    user control subroutines, 9-15, 9-16
    variables, 9-13
Name
    array, 4-13
    common block, 4-32, 4-38
    symbolic, 2-7
Natural log, intrinsic function, B-7
.NE. operator, 5-15, 5-18
NEQV, intrinsic function, B-8
NEW, file status, 7-30
NEXTREC=, INQUIRE specifier, 7-34
NINT, intrinsic function, B-2
NO SIDE EFFECTS directive, 1-30
NOBL directive, 1-32
NOEMA designation, 1-26
NOLIST directive, 1-5, 1-18
Non-Fortran procedures, 2-26, F-1
    CAL, F-1
    Cray C, F-2
    Cray Pascal, F-1
    functions, 2-21
    subroutines, 2-21
Nondefault dataset, 1-14
Nonexecutable statement, 2-4
Normalized values, 3-7

Novector, optimization option, 1-8, 1-22
NULL, blank specifier, 7-30
NUMARG, intrinsic function, B-10
NUMBER=, INQUIRE specifier, 7-34
Numeric storage unit, 4-28

-o parameter (optimization), cft77 command, 1-8
O format descriptor (octal), 8-24
o/O compiler option (bounds checking), 1-11 (UNICOS), 1-24 (COS), 1-34, 4-21
Octal format descriptor (O), 8-24
OFF= parameter (disables options under COS), 1-21
Off, optimization option, 1-8, 1-22
OLD, file status, 7-30
ON parameter, CFT77 statement, 1-22, 1-24
OPEN statement, 7-28
    access specifier, 7-36
    alternatives to, 7-29
    creating files, 7-11
    example, 7-1
    file identifier, 7-13
OPEN statement (continued)
    specifiers (table), 7-30
    unit specifier, 7-16
OPENED=, INQUIRE specifier, 7-34
Operands, arithmetic, 5-6, 5-1
Operators
    arithmetic, 5-5
    contatenation (//), 5-13
    defined, 2-2
    in expression, 5-1, 5-5 (table)
    logical and masking, 5-22
OPT parameter, CFT77 statement, 1-22
Optimizing, 10-1
    bottom-loading, 1-32
    command options, 1-8 (UNICOS), 1-22 (COS)
    directives, 1-31
    multitasking, 10-3
    options, 1-22
    and pointers, 3-19
    status, 1-37
    suppressing (SUPPRESS directive), 1-31
    vectorization, 10-4
Optionally signed constant, 4-1
Options, compiler, 1-10 (UNICOS), 1-24 (COS)
    enabling, 1-7 (UNICOS), 1-20 (COS)
    local control, 1-33
OR, intrinsic function, B-8
Order of statements and lines, 2-5
Output to a printer, 7-26
$OUT dataset, 1-18, 1-13, 1-21, 1-24, 7-17
    and namelist processing, 9-14

P format descriptor (scale factor), 8-20
p/P compiler option, 1-10, 1-12 (UNICOS), 1-24 (COS)
Page of source listing, 1-33
    header lines, 1-37

PARAMETER statement, 4-1, 2-6
    and character constant, 3-12
    dummy argument not in, 2-34
    in block data subprogram, 4-45
Parameter table, 1-41
Parentheses in expressions, 5-1
Partial association, 4-33
Pascal, procedures in, F-1
PAUSE statement, 6-15
Plus sign format descriptor (S, SS, SP), 8-30
Pointer type, 3-16
        and SAVE statement, 4-31
        array, 3-17, 4-5
        in DATA statement, 4-28
        listing, 1-40
        not dummy argument, 2-34
    POINTER type statement, 3-17
        and entry name, 2-29
        for array, 4-5
    using pointers, 3-18
POPARR, intrinsic function, B-9
POPCNT, intrinsic function, B-9
Population count, intrinsic function, B-9
Population parity, intrinsic function, B-9
Position format (T, TL, TR, X), 8-28
Powers and constants, C-1
Precedence of all operators, 5-1
Prefetching operands, 1-32
Primaries, in arithmetic expressions, 5-6
Print control characters (table), 7-26
PRINT statement, see Data transfer, 7-19, 1-16
    unit specifier, 7-16
    with namelist I/O, 9-10
Printer, output to, 7-26
Procedure
    call, 2-8
    dummy, 2-35, 2-7, 2-27, 3-4
    subprogram, 2-8, 2-9
Product, logical, intrinsic function, B-8
Program control, 6-1
    conditional blocks, 6-1
    DO loops, 6-6
    GOTO and ASSIGN, 6-11
    IF statement, 6-4
    suspending execution, 6-14
Program example, 2-10
    using NAMELIST, 9-11
PROGRAM statement, 2-12, 2-6
    with Flowtrace, 1-34
Program structure, 2-8
    arguments, 2-32
    function, 2-13
    program unit, 2-11, 2-8
    subprogram, 2-21
    example, 2-10
    summary, 2-9
PUNCH statement, E-11
    with namelist I/O, 9-10
$PUNCH, 7-17
Pure data file, 7-12
PUTWA, 9-5

q/Q compiler option, 1-12 (UNICOS), 1-25 (COS)

R format descriptor (right-justified), E-8
r/R compiler option, 1-12 (UNICOS) 1-25 (COS)
Random I/O, 9-5, 7-12
    comparison, 9-6 (table)
    characteristics of (table), 9-7
Random number, intrinsic function, B-3
RANF, intrinsic function, B-3
Range, real type, 3-7
RANGET, intrinsic function, B-3
RANSET, intrinsic function, B-3
$RBUFLN, 7-40
READ statement, see Data transfer, 7-19
    and end-of-file, 7-21
    file identifier, 7-13
    unit specifier, 7-16, 7-21
    with namelist I/O, 9-10
READDR, 9-5
READMS, 9-5
REAL, intrinsic function, 2-37, B-4
Real type, 3-7
    conversion, intrinsic function, B-4
    conversion to and from, 5-12
    formatting, 8-13
        input (F, E, G), 8-17
        output, 8-14 (F), 8-15 (E), 8-16 (G)
    machine representation-G-2
    normalized values, G-3
    operation, with Boolean item, 5-10
    type statement, 3-4
Real-time clock, intrinsic function, B-10
REC= specifier, I/O statement, 7-21
RECL=, INQUIRE specifier, 7-34
RECL=, OPEN specifier, 7-30
Record, 7-10, 7-21
    blocking, 7-10
    length specifier, OPEN statement, 7-30
    specifier, INQUIRE, 7-34
Recurrence, in vectorizing, 10-4
Recursion, 2-17, 2-22, 2-29
    and dummy procedures, 2-36
    and SAVE statement, 1-31
    of function subprogram, 2-23
Redirection of I/O files, 7-17
Reference (to a constant, etc.), 2-7
Relational expressions, 5-15
    arithmetic, 5-16
    array section in, 4-21
    character, 5-17
    data types in (table), 5-18
Remainder, divide for (MOD), intrinsic function, B-2
Representation of data, machine, G-1
Result variable, function, 2-14, 2-23
RETURN statement, 2-21, 2-23, 2-26, 2-30, 6-7
    storing pointee variable, 3-19
    timing with Flowtrace, 1-34
REWIND statement, 7-38
    file identifier, 7-13

$RFDCOM, common blocks, 7-40
Right-justified character specifier (R), E-8
RNL routines, 9-15
RNLTYPE routines, 9-15
Rounding, 3-7, 3-8
    multiply operations, 1-12
RTC, intrinsic function, B-10

-s parameter (CAL file), cft77 command, 1-9
S, SS format descriptor (suppress + sign), 8-30
s/S compiler option (listings)
    UNICOS, 1-3, 1-4, 1-5, 1-11, 1-12, 1-37
    COS, 1-18, 1-25, 1-37
SAVE command (COS datasets), 7-11, 7-28
SAVE statement, 4-31, 1-4, 1-17
    block data subprogram, 4-45
    common block name in, 4-40
    definition of entities in, 2-21
    dummy argument not in, 2-34
    and local common, 4-44
    and pointees, 3-17
    static allocation, 1-6
    static variables, 4-3
    storage, 4-29
Scalar item, in array syntax, 4-21
Scalar merge, intrinsic function, B-9
Scalar optimization, 1-31
Scalar variable, 4-3
Scale factor format descriptor (P), 8-20
Scope of a symbolic name, 2-7
SCRATCH, file status, 7-30
Section selectors, array, 4-16
SEEK, with GETWA routine, 9-5
SEGLDR
    and common block size, 4-42
    control statement, 1-15
    directives
        changing I/O buffers, 7-40
        load map, 1-4, 1-16
        loading UNICOS libraries (**prof** and Perftrace), 10-2, 10-3
        renaming program for running, 10-2
    statement (COS), 1-14
    command (UNICOS), 1-2
    and stack storage, 4-29
Sequence, storage, 4-28
Sequential file access, also see File access, 7-36
    file status, 7-30
    INQUIRE specifier, 7-34
SET, SEGLDR directive, 7-40
SETPLIMQ subroutine, Flowtrace▸, 1-34
SETPOS, 9-5
    with BUFFER IN/OUT, 9-7
    with buffered I/O, 9-1
Shape, of array operands, 4-21
SHIFTL/R, intrinsic functions, B-9
SHORTLOOP directive, 1-30
Side effects, in vectorization, 1-30
Sign transfer, intrinsic function, B-2
Signed constant, 4-1
Simple variable, 4-3
SIN, SINH, intrinsic function, B-6
SNGL, intrinsic function, 2-37
SNGL, intrinsic function, B-4
Source code
    listing directives, 1-33
    listing, 1-37
        UNICOS, 1-2, 1-3, 1-5, 1-7, 1-10, 1-11, 1-12
        COS, 1-13, 1-14, 1-16, 1-21, 1-24, 1-25
    INCLUDE files, 1-36
Source program references, 1-41
SP format descriptor (include + sign), 8-30
Special character, 2-1
Specification statement, 2-5, 2-6
    and DATA statement, 4-23
Specification subprogram, 2-8
Specifier, 7-35
SQRT, intrinsic function, B-7
Square root, intrinsic function, B-7
STACK directive, SEGLDR control statement, 4-29
STACK option, 1-19
Stack storage, 4-29, 4-43
    and association, 4-34
    and DATA statement, 4-23
    listing, 1-40
    mode, 1-6, 2-22
    variable, 4-3
STANDARD parameter, CFT77 statement, 1-23
Statement function, 2-16, 2-13, 2-8
    definition statement, 2-17
    Type specification, 2-19
Statements, 2-4
    labels, 2-2
        and numbers, 1-37, 1-41
        for alternate return, 2-31
    order, 2-5, 2-6
STATIC option, 1-19
Static storage, 4-29, 4-43
    and association, 4-34
    and SAVE statement, 4-31
    listing, 1-40
    mode, 1-6, 2-22
    variable, storage allocation, 4-3
Status identifier, 7-21
STATUS=, CLOSE specifier, 7-31
STATUS=, OPEN specifier, 7-30
stderr UNICOS file, 7-17
stdin UNICOS file, 1-2, 7-17
stdout UNICOS file, 1-2, 1-3, 7-17
STOP statement, 6-14, 2-23, 6-7
Storage, 4-28
    absolute locations, 3-17
    allocation, 1-4, 1-6, 1-19, 3-16
        directive, 1-35
        heap, 1-6, 1-19, 1-30, D-1
        static variable, 4-3
    array
        sequence (figure), 4-10, 4-11
    and association, 4-28, 4-32
    and data structures, 4-1

Storage (continued)
heap, 1-6, 1-19, 1-30, D-1
listing, 1-40
memory paging (CRAY-2), 1-10, 1-12
register, and subprogram calls, 1-20
sequence, 4-28
and association, 4-28, 4-32
array, 4-10, 4-11
common block, 4-28, 4-40
of an array (figure), 4-15
stack and static, 4-29, 1-6, 2-22, 4-43
unit, 4-28, 5-22
and arrays, 4-12
character, 4-28
with pointers, 3-16, 3-19
String length, intrinsic function, B-5
Striping, random I/O, 9-6
Subprogram, 2-21, 2-8, 4-9
altering transfer of control, 2-28
ENTRY statement, 2-29
RETURN statement, 2-30
argument, 2-32
array in, 4-5
external functions, 2-22
restrictions, 2-22
FUNCTION statement, 2-24
function subprograms, 2-22
name, 2-7
as argument, 2-32, 2-34
referencing itself, 2-22, 4-31
side effects, 1-30
stack storage, 4-29
subroutines, 2-25, 2-26, 2-8, 2-9
CALL statement, 2-26
requirements, 2-26
SUBROUTINE statement, 2-28
tracing, 1-33
SUBROUTINE statement, 2-28, 2-6, 2-21, 2-22, 2-26
alternate return specifier in, 2-31
argument used in statement function, 2-18
Subscript expressions, array, 4-9
value in argument, 2-33
Subscripted variable (term), 4-3
Substring, character, 3-13, 3-14
of array element, 4-12
of array, 4-4
index, intrinsic function, B-5
name as argument, 2-32, 2-33
name in implied-DO list, 4-26
name in EQUIVALENCE statement, 4-34
Sum, logical, intrinsic function, B-8
Suppressing optimization (SUPPRESS), 1-31
Symbol Cross-reference Table, 1-38
enabling, 1-10, 1-12, 1-23, 1-25
Symbolic constant, 4-1
Symbolic names, 2-2, 2-7, 3-4
data type of, 3-3
number of characters, 1-1, 2-7
SYNCDR, 9-5
Syntactic items, 2-2

-t cft77 command parameter, 1-9
T format descriptor (position), 8-28
TAN, TANH, intrinsic functions, B-6
TARGET command (COS), 1-20
target command (UNICOS), 1-6
Task common, entity in DATA statement, 4-28
TASK COMMON statement, 4-43, 4-38
variables, 4-3
TASK common, 1-30, 1-6
Term, in arithmetic expression, 5-7
Terminal line, 2-3, 6-15
Terminal statement of DO loop, 6-8, 6-6, 6-7
Text file, 7-12
Time functions (table), B-10
Time of compilation, 1-37
TL format descriptor (backward), 8-28
Total association, 4-32
TR format descriptor (forward), 8-28
Transferring file to front-end computer, 7-2
Trigonometric functions (table), B-6
Trip count, 6-6, 6-8, 6-10
of implied-DO (I/O), 7-24
TRUNC parameter, CFT77 statement, 1-22
Truncation, 3-7, 3-8
in expressions, 5-13
parameter, 1-9 (UNICOS), 1-22 (COS)
in listing, 1-37
Types, data, 3-1; also see Data types

Unblocked datasets, with BUFFER IN/OUT, 9-1
Undefined variable, 4-30
UNFORMATTED, file status, 7-30
Unformatted I/O, 8-1
I/O statements, 7-10
tutorial, 7-3
Unformatted record, 7-10
UNFORMATTED=, INQUIRE specifier, 7-35
UNICOS, compiling under, 1-2
UNICOS, range of units, 7-16
Uninitialized variable, 1-10, 1-23
Unit, I/O, 7-20
Unit, storage, 4-28
UNIT function, 9-3, 9-1
UNIT= identifier, I/O statements, 7-20, 7-30, 7-31
Unit, I/O, 7-16
assigned in OPEN statement, 7-28
creation and assignment, 7-11
in INQUIRE statement, 7-33
in INQUIRE statement, 7-34
UNKNOWN, file status, 7-30
Unsigned constant, 4-1
US parameter, JOB statement, 1-13

-V option, cft77 command, 1-2
Variable, 4-3
name in type statement, 3-4
name, 2-7
pointer, 3-16
uninitialized, 1-10
VECTOR and NOVECTOR directive, 1-28

Vectorization, 10-4
    and array bounds checking, 1-34
    command options, 1-8 (UNICOS),
      1-22 (COS)
    dependencies, 1-28
    directives, 1-28
    enabling listings, 1-10
    expressions, 10-5
    loops containing IF, 10-5
    loops, 10-4
    options, 1-22
    recurrences, 10-6
    statements, 10-4
    suppressing, 1-28
Vectorization control directives, 1-28
    IVDEP, 1-28
    NO SIDE EFFECTS, 1-30
    SHORTLOOP, 1-30
    VECTOR and NOVECTOR, 1-28
    VFUNCTION, 1-29
Vector array reference, 10-4
Vector merge functions (table), B-11
Vector population count, 1-20
VFUNCTION directive, 1-29, 10-5
vpop/VPOP option, 1-7 (UNICOS), 1-20 (COS)

w/W compiler option, 1-12 (UNICOS)
WAITDR, 9-5
$WBUFLN, 7-40
$WFDCOM, common blocks, 7-40
Whole array reference, 4-20
Whole number, nearest, intrinsic function,
  B-2
WNL routines, 9-17
WNLDELM routines, 9-17
WRITDR, 9-5
    Also see Data transfer
WRITE statement, 1-16, 7-19
    file identifier, 7-13
    unit identifier, 7-20, 7-16
WRITE statement (continued)
    vectorizing, 10-4
    with namelist I/O, 9-10
WRITMS, 9-5
WWAIT function, 9-5

X format descriptor (forward), 8-28
x/X compiler option
    UNICOS, 1-12, 1-2, 1-3, 1-5
    COS, 1-25, 1-13, 1-16, 1-18
XOR, intrinsic function, B-8

ZERO, blank specifier, 7-30
zeroinc optimization option, 1-8
ZEROINC, optimization option (COS) 1-22

READER'S COMMENT FORM

CFT77 Reference Manual SR-0018

Your reactions to this manual will help us provide you with better documentation. Please take a moment t check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
   ____ other (please specify): ____________________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
   ____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

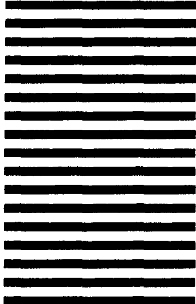
Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ______________________
Title ______________________
Company ______________________
Telephone ______________________
Today's Date ______________________

Address ______________________
City ______________________
State/ Country ______________________
Zip Code ______________________

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

READER'S COMMENT FORM

CFT77 Reference Manual SR-0018

Your reactions to this manual will help us provide you with better documentation. Please take a moment check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
   ____ other (please specify): ___________________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
   ____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

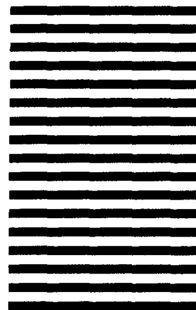
Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ________________________
Title ________________________
Company ________________________
Telephone ________________________
Today's Date ________________________

Address ________________________
City ________________________
State/ Country ________________________
Zip Code ________________________

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120