CONVEX FORTRAN Language Reference Manual

Document No. 720-000050-203

Seventh Edition
October 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX FORTRAN Language Reference Manual
Order No. DSW-037
Seventh Edition

© 1986, 1987, 1988 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, stored electronically, or reduced to machine-readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation (CONVEX) does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
COVUE is a trademark of CONVEX Computer Corporation.
Cray is a registered trademark of Cray Research, Inc.
Sun FORTRAN is a trademark of Sun Microsystems, Inc.
UNIX is a trademark of AT&T Bell Laboratories.
VAX and VMS are trademarks of Digital Equipment Corporation.

Printed in the United States of America

Revision Information for
CONVEX FORTRAN Language Reference Manual

Edition	Document No.	Description
Seventh	720-000050-203	Released with CONVEX FORTRAN software V5.0, November 1988. Added descriptions of new directives SYNCH_PARALLEL, FORCE_VECTOR, FORCE_PARALLEL, BEGIN_TASKS, NEXT_TASK, END_TASKS, PSTRIP, VSTRIP. Added new command line options -O3, -or, -ep, -re, -tm; deleted option -fx. Added description of variable formats. Removed Chapter 9, combined material on main programs into other chapters, and renumbered Chapters 10 and 11. Removed Appendix A, relocated preprocessor statements to Appendix J, added system limits as Appendix D, and renumbered other appendixes as necessary. Expanded preprocessor appendix to contain all preprocessor information. Corrected all reported errors and performed major editorial revisions.
Sixth Rev. 1	720-000050-202	Released with CONVEX FORTRAN software V4.1, May 1988. Added descriptions of new options -fpp, -rl, -sfc, -vfc. Added descriptions of new directives UNROLL, SELECT, MAX_TRIPS, ROW_WISE. Changed option -cft to -cfc. Added new VMS FORTRAN features to Appendix I. Added Appendix J to describe Sun FORTRAN compatibility and moved problem reporting to Appendix K.
6.0	720-000050-201	Released with CONVEX FORTRAN software V4.0, November 1987. Added IEEE, Cray, and inline substitution options. Changed Appendix H to describe Cray compatibility; revised VAX compatibility appendix and made it Appendix I. Moved problem reporting to Appendix J.
5.0	720-000050-200	Released with CONVEX FORTRAN software V3.0, May 1987. Updated entire manual to reflect enhancements for CONVEX FORTRAN V3.0 and to correct errors.
4.0	720-000099-000	Released with CONVEX FORTRAN software V2.2, September 1986. Added NAMELIST format information to Chapter 3. In Appendix C, changed or added information on options -B, -On, -U, -db, -g, -kb, and -sc.
3.0	720-000150-000	Released with CONVEX FORTRAN software V2.0, April 1986.
2.0	720-000150-000	Released with CONVEX FORTRAN software V1.7, September 1985.
1.0	720-000150-100	Released with CONVEX FORTRAN software V1.0, February 1985. First release of the manual

3.5.0.4 Equivalencing Substrings lrm-3-5
3.5.0.5 Using EQUIVALENCE in Common Blocks lrm-3-6
3.6 PARAMETER Statement lrm-3-6
3.6.1 Standard PARAMETER statement lrm-3-6
3.6.2 Alternate PARAMETER Statement lrm-3-7
3.7 PROGRAM Statement lrm-3-7
3.8 NAMELIST Statement lrm-3-8
3.9 EXTERNAL Statement lrm-3-9
3.10 INTRINSIC Statement lrm-3-9
3.11 SAVE Statement lrm-3-9

4 DATA Statement
4.1 DATA Statement Form lrm-4-1
4.1.1 Implied-DO lrm-4-2
4.2 DATA Statement Extensions lrm-4-3

5 Assignment Statements
5.1 Character Conversions lrm-5-1
5.2 ASSIGN Statement lrm-5-3

6 Control Statements
6.1 GOTO Statements lrm-6-1
6.1.1 Unconditional GOTO Statement lrm-6-1
6.1.2 Computed GOTO Statement lrm-6-2
6.1.3 Assigned GOTO Statement lrm-6-2
6.2 IF Statements lrm-6-3
6.2.1 Arithmetic IF Statement lrm-6-3
6.2.2 Logical IF Statement lrm-6-3
6.2.3 Block IF Statement lrm-6-4
6.2.3.1 Nested Block IF Statements lrm-6-6
6.3 DO Statements lrm-6-6
6.3.0.2 Nested DO Loops lrm-6-7
6.3.0.3 Extended Range lrm-6-8
6.3.1 DO WHILE Statement lrm-6-8
6.4 END DO Statement lrm-6-9
6.5 CONTINUE Statement lrm-6-9
6.6 STOP Statement lrm-6-10
6.7 PAUSE Statement lrm-6-10
6.8 END Statement lrm-6-10

7 Input/Output Statements
7.1 Records lrm-7-1
7.1.1 Formatted Records lrm-7-2
7.1.2 Unformatted Records lrm-7-2
7.1.3 ENDFILE Record lrm-7-2
7.2 Files lrm-7-2
7.2.1 Internal Files lrm-7-2
7.3 Units lrm-7-3
7.4 Accessing Files lrm-7-3
7.4.1 Sequential Access lrm-7-3
7.4.2 Direct Access lrm-7-4
7.5 I/O Statement Format lrm-7-4
7.5.1 Input/Output Lists lrm-7-4
7.5.2 Implied-DO Lists lrm-7-5
7.6 Specifiers lrm-7-5
7.6.1 Unit Specifier lrm-7-6
7.6.2 Format Specifier lrm-7-6
7.6.3 Record Specifier lrm-7-6

7.6.5 Error Specifier lrm-7-7
7.6.6 End-of-File Specifier lrm-7-7
7.6.7 Namelist Specifier lrm-7-8
7.7 READ Statement lrm-7-8
7.7.1 External Sequential READ Statements lrm-7-9
7.7.2 External Direct READ Statements lrm-7-10
7.7.3 Internal READ Statements lrm-7-11
7.8 ACCEPT Statement lrm-7-11
7.9 WRITE Statement lrm-7-12
7.9.1 Sequential-Access WRITE Statements lrm-7-12
7.9.2 Direct-Access WRITE Statements lrm-7-13
7.9.3 Internal Direct-Access WRITE Statements lrm-7-14
7.10 PRINT and TYPE Statements lrm-7-14
7.11 Additional Statements lrm-7-15
7.11.1 ENCODE Statement lrm-7-15
7.11.2 DECODE Statement lrm-7-16
7.11.3 FIND Statement lrm-7-17
7.12 Auxiliary Input/Output Statements lrm-7-18
7.12.1 OPEN Statement lrm-7-18
7.12.1.1 ACCESS Keyword lrm-7-18
7.12.1.2 ASSOCIATEVARIABLE Keyword lrm-7-20
7.12.1.3 BLANK Keyword lrm-7-20
7.12.1.4 BLOCKSIZE Keyword lrm-7-20
7.12.1.5 CARRIAGECONTROL Keyword lrm-7-20
7.12.1.6 DEFAULTFILE Keyword lrm-7-21
7.12.1.7 DISPOSE Keyword lrm-7-21
7.12.1.8 ERR Keyword lrm-7-21
7.12.1.9 FILE Keyword lrm-7-21
7.12.1.10 FORM Keyword lrm-7-22
7.12.1.11 IOSTAT Keyword lrm-7-22
7.12.1.12 MAXREC Keyword lrm-7-22
7.12.1.13 NOSPANBLOCKS Keyword lrm-7-23
7.12.1.14 READONLY Keyword lrm-7-23
7.12.1.15 RECL Keyword lrm-7-23
7.12.1.16 RECORDTYPE Keyword lrm-7-23
7.12.1.17 STATUS Keyword lrm-7-24
7.12.1.18 UNIT Keyword lrm-7-24
7.12.2 CLOSE Statement lrm-7-25
7.12.3 INQUIRE Statement lrm-7-25
7.12.4 File-Positioning Statements lrm-7-28
7.12.4.1 REWIND Statement lrm-7-28
7.12.4.2 BACKSPACE Statement lrm-7-28
7.12.4.3 ENDFILE Statement lrm-7-29

8 Format Specifications
8.1 FORMAT Statement lrm-8-1
8.2 FORMAT Control lrm-8-2
8.2.1 Repeat Count lrm-8-3
8.3 Descriptors lrm-8-3
8.3.1 A Descriptor lrm-8-4
8.3.2 Apostrophe Descriptor lrm-8-5
8.3.3 H Descriptor lrm-8-5
8.3.4 L Descriptor lrm-8-5
8.3.5 I Descriptor lrm-8-6
8.3.6 O Descriptor lrm-8-7
8.3.7 Z Descriptor lrm-8-7
8.3.8 F Descriptor lrm-8-8
8.3.9 E and D Descriptors lrm-8-9
8.3.10 G Descriptor lrm-8-10

8.3.11 B Descriptors lrm-8-12
8.3.12 P Descriptor lrm-8-13
8.3.13 S Descriptors lrm-8-13
8.3.14 R Descriptor lrm-8-14
8.3.15 X Descriptor lrm-8-14
8.3.16 T Descriptors lrm-8-15
8.3.17 $ Descriptor lrm-8-16
8.3.18 Q Descriptor lrm-8-16
8.3.19 Colon Descriptor lrm-8-17
8.3.20 Slash Descriptor lrm-8-17
8.4 Default Field Descriptor Values lrm-8-17
8.5 Comma Field Separator on Input Data lrm-8-18
8.6 Runtime Formats lrm-8-19
8.7 Variable Formats lrm-8-19
8.8 List-Directed Formatting lrm-8-20
8.8.1 Input lrm-8-20
8.8.1.1 Character Input lrm-8-20
8.8.1.2 Nulls and Slashes lrm-8-21
8.8.2 Namelist-Directed Formatting lrm-8-21
8.8.3 List-Directed Output lrm-8-23
8.8.4 Namelist-Directed Output lrm-8-23
8.9 Carriage-Control Characters lrm-8-24

9 Subprograms
9.1 Dummy and Actual Arguments lrm-9-1
9.1.1 Variables as Dummy Arguments lrm-9-1
9.1.2 Arrays as Dummy Arguments lrm-9-2
9.1.2.1 Adjustable Arrays lrm-9-2
9.1.2.2 Assumed-Size Arrays lrm-9-3
9.1.3 Character Arguments lrm-9-3
9.1.3.1 Character Argument Lengths lrm-9-3
9.1.4 Procedures as Dummy Arguments lrm-9-4
9.1.5 Alternate Return Arguments lrm-9-5
9.2 Functions lrm-9-5
9.2.1 Intrinsic Functions lrm-9-5
9.2.2 Built-in Functions lrm-9-6
9.2.2.1 %REF and %VAL lrm-9-6
9.2.2.2 %LOC lrm-9-7
9.2.3 Statement Functions lrm-9-7
9.3 Function Subprograms lrm-9-9
9.4 Subroutine Subprograms lrm-9-10
9.5 ENTRY Statement lrm-9-11
9.6 RETURN Statement lrm-9-12

10 Block Data Subprogram

Appendices

A Generic and Intrinsic Functions A-1
B Compiler Options B-1
B.1 Compiling Programs lrm-B-1
C Compiler Directives C-1
C.1 Information Directives lrm-C-1
C.2 Control Directives lrm-C-3
D System Limits D-1
E ASCII Character Set E-1
F FORTRAN-66 Compatibility F-1

F.1 Compiling FORTRAN-66 Programs ... lrm-F-1
F.2 EXTERNAL Statement ... lrm-F-1
F.3 DO Loop Minimum Iteration Count ... lrm-F-2
F.4 OPEN Statement Keywords ... lrm-F-2
F.5 X Format Edit Descriptor ... lrm-F-3
G Cray FORTRAN Compatibility ... G-1
H VAX FORTRAN Compatibility ... H-1
H.1 Supported Features ... lrm-H-1
H.2 Unsupported Features ... lrm-H-1
H.3 Miscellaneous Differences ... lrm-H-3
H.4 VAX FORTRAN Records ... lrm-H-3
I Sun FORTRAN Compatibility ... I-1
J Preprocessor ... J-1
J.1 Preprocessor Statements ... lrm-J-1
J.2 Preprocessor Options ... lrm-J-2
J.3 Preprocessor Messages ... lrm-J-2
K Problem Reporting ... K-1
K.1 Introduction ... lrm-K-1
K.2 Information Required to Report a Problem ... lrm-K-1

List of Tables

1-1 FORTRAN Fields ... lrm-1-3
1-2 OPTIONS Statement ... lrm-1-6
2-1 Data Types ... lrm-2-1
2-2 Storage Requirements for Data Types ... lrm-2-2
3-1 Array Locations ... lrm-3-5
5-1 Conversion of Expressions ... lrm-5-2
7-1 Data Transfer I/O Statements ... lrm-7-1
7-2 OPEN Statement Keywords ... lrm-7-19
7-3 INQUIRE Specifiers ... lrm-7-27
8-1 Character Assignment for Numeric I/O List Elements ... lrm-8-4
8-2 Default Field Descriptors ... lrm-8-18
8-3 List-Directed Output Formats ... lrm-8-23
8-4 Vertical Format Control ... lrm-8-25
9-1 Built-in Functions and Defaults for Argument Lists ... lrm-9-7
A-1 Generic and Intrinsic Functions ... lrm-A-1
C-1 Restrictions on Directive Use ... lrm-C-4
C-2 Maximum Strip Mine Lengths ... lrm-C-6
E-1 ASCII Character Set ... lrm-E-1

List of Figures

1-1 Required Order of Statements ... lrm-1-5
6-1 Valid and Invalid Control Transfers ... lrm-6-8
K-1 Sample *contact* Session ... lrm-K-3

Preface

This manual is a reference for the CONVEX FORTRAN programming language. Its purpose is to provide a thorough working definition of CONVEX FORTRAN. This document encompasses both the ANSI standard established by the American National Standards Institute (ANSI X3.9-1978) and the CONVEX extensions to FORTRAN-77.

This manual is intended for use by those individuals who are responsible for developing FORTRAN programs for the CONVEX family of supercomputers. A working knowledge of FORTRAN-77 is a prerequisite to the use of this manual.

As this document is intended for the primary use of the seasoned programmer, a novice programmer may require additional reference materials. Consult the bibliography at the end of the Preface for a listing of useful texts that supplement the material contained in this document. Although you do not need to know the CONVEX version of the UNIX operating system in detail to understand this document, some familiarity with the system is beneficial.

Organization

This manual is organized into the following chapters and appendices:

- Chapter 1 discusses FORTRAN program elements and program unit format.
- Chapter 2 discusses constants, variables, arrays, expressions, and function references.
- Chapter 3 describes features and functions of specification statements.
- Chapter 4 discusses the DATA statement.
- Chapter 5 describes the assignment statement and defines values used in a program.
- Chapter 6 describes the functions and operations of control statements.
- Chapter 7 discusses files, units, I/O statement components, data transfer I/O statements, and auxiliary I/O statements.
- Chapter 8 defines the format specification descriptors as well as carriage-control options and separators.
- Chapter 9 discusses the functions and operations of subprograms.
- Chapter 10 discusses the BLOCK DATA statement.
- Appendix A lists the generic and intrinsic functions in table form, complete with number of arguments, generic name, specific name, type of argument, and type of result.
- Appendix B describes the compiler options.
- Appendix C lists the compiler directives.
- Appendix D lists the maximum sizes for the various elements in a FORTRAN program.
- Appendix E contains the FORTRAN and ASCII character sets.
- Appendix F discusses FORTRAN-66 compatibility.

- Appendix G discusses Cray FORTRAN compatibility.
- Appendix H discusses VAX FORTRAN compatibility.
- Appendix I discusses Sun FORTRAN compatibility.
- Appendix J describes the CONVEX FORTRAN preprocessor.
- Appendix K provides information for using the *contact* problem reporting utility.

An index and reader reply forms are included at the back of the manual.

Notational Conventions

The following conventions are used in this document:

- Brackets ([ ]) designate optional entries.
- The ^ mark represents the nonprinting space character.
- A horizontal ellipsis (...) shows repetition of the preceding item(s).
- A vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- References to the *CONVEX UNIX Programmer's Manual* appear in the form *fc*(1F), where the name of the manual page is followed by its section number enclosed in parentheses.
- Within text, *italics* indicate commands, options, filenames, or programs.
- Within command sequences, words appearing in **boldface** should be typed exactly as they appear; words appearing in *italics* indicate items to be filled in by the programmer. Substitute the actual name of the item for the word in italics. For example, the command sequence

  ld [*options*] [*object files*] [*libraries*]

  instructs you to type the command *ld*, followed by your choice of options, object files, and/or libraries.
- *CONVEX extensions to the FORTRAN language appear in this type style.*

Associated Documents

The following documents, available from CONVEX Computer Corporation, are recommended to the CONVEX FORTRAN programmer:

- *CONVEX FORTRAN User's Guide*, describes how to compile and run CONVEX FORTRAN programs under the UNIX operating system.
- *CONVEX UNIX Primer*, contains basic self-instruction for learning and using the CONVEX UNIX operating system.
- *CONVEX UNIX Programmer's Manual*, Parts I and II, contains complete reference material on the UNIX operating system for the CONVEX family of supercomputers.
- *American National Standard Programming Language, FORTRAN* manual (ANSI X3.9-1978) defines the standard language.

For further reference, Section 3F of the *CONVEX UNIX Programmer's Manual* contains the FORTRAN runtime library functions.

Chapter 1
Introduction

CONVEX FORTRAN is a high-level language that increases programmer productivity, maximizes software portability, and enhances the speed of execution using global and local optimization, vectorization, and parallelization techniques. CONVEX FORTRAN contains both standard FORTRAN functions as defined by the American National Standard FORTRAN-77 (ANSI X3.9-1978) and unique CONVEX extensions. ***This sentence illustrates the type style that is used to indicate CONVEX extensions throughout the document.***

1.1 Program Elements

A CONVEX FORTRAN program unit consists of a sequence of FORTRAN statements, an optional ***OPTIONS statement,*** and optional comment lines. A program unit terminates with an END statement.

Collectively, program units build an executable program. An executable program contains a main program and, optionally, one or more subprograms. The first statement of a main program cannot be a FUNCTION, SUBROUTINE, or BLOCK DATA statement. A main program may, but need not, begin with a PROGRAM statement. A subprogram begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

1.1.1 FORTRAN Character Set

If you are entering your CONVEX FORTRAN program from a terminal, you may enter statement lines of any length as long as you do not exceed the system limit (see Appendix D) or use the -72 compiler option. Statements are coded using the conventional FORTRAN character set:

Uppercase letters	A through Z
Digits	0 through 9
Special characters	blank = + - * / ( ) , . $ ' :

and the following CONVEX extensions:

Lowercase letters	*a - z*
Exclamation mark	*!*
Percent sign	*%*
Ampersand	*&*
Quotation mark	*"*
Underscore	*_*
Left angle bracket	*<*
Right angle bracket	*>*
Pound sign	*#*
Semicolon	*;*

Additional ASCII printable characters can be used in FORTRAN statements only as part of character or Hollerith constants. You can, however, use all printable ASCII characters in comment lines.

Blanks (spaces) can be used to improve readability of a program. Blanks are ignored unless they appear within a character or a Hollerith constant or as an editing specification.

1.1.2 Comment Line

A comment line has no effect on the actual execution of a program. It is used for documenting program action, identifying processes, or improving readability of the program. You can place a comment line anywhere in a program unit, even before the initial line or between continuation lines. You cannot continue a comment line using the continuation indicator.

The letter C or an asterisk (*) in column 1 of a line indicates a comment line. ***Also, an exclamation point (!) in column 1 (or any field except column 6) indicates that the remainder of the line is comment text.*** You can begin the comment text anywhere on the line following the comment indicator. A line containing only blanks is also a comment line.

1.2 Building FORTRAN Statements

Statements are classified as executable or nonexecutable. The executable statements specify action; they form an execution sequence in an executable program. The nonexecutable statements indicate characteristics, arrangement, and initial values of variables; contain editing information; classify program units; and designate entry points within subprograms.

Statements classified as executable include:

- Arithmetic, logical, statement label (ASSIGN), and character assignment statements
- Unconditional GOTO, assigned GOTO, and computed GOTO statements
- Arithmetic IF and logical IF statements
- Block IF, ELSE IF, ELSE, and END IF statements
- CONTINUE statement
- STOP and PAUSE statements
- DO ***and ENDDO*** statements
- READ, WRITE, and PRINT statements
- REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE statements
- CALL and RETURN statements
- END statement

Statements classified as nonexecutable include:

- PROGRAM, FUNCTION, SUBROUTINE, ENTRY, and BLOCK DATA statements
- DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC, and SAVE statements
- INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER statements
- DATA statement
- FORMAT statement
- Statement function statement

Each line that contains a FORTRAN statement is divided into fields providing for statement label, continuation indicator, statement text, and a sequence number. Table 1-1 gives the general rules for entering items into fields.

You can use either character-per-column or ***tab-character formatting. Tab formatting is convenient for terminal entry.***

Table 1-1: FORTRAN Fields

Column	Use
1 (Comment or debugger)	The letter C, *exclamation point*, or asterisk indicates a comment line. *The letter D designates a debugging statement.*
1 - 5 (Statement label or compiler directive)	A statement label contains one to five digits. *C$DIR indicates a compiler directive.*
6 (Initial or continuation line)	A zero or blank indicates an initial line; any other character indicates a continuation line; *for tab formatting, a tab and a digit 1 - 9 indicate continuation.*
7 - end of line (Statements)	This field contains the actual FORTRAN statement. *At any point in the field, an exclamation point indicates that the remaining text is a comment.*

1.2.1 Character-per-Column Formatting

This section describes the column-by column contents of each field of a FORTRAN statement. Optional information that can be entered in the various fields is also discussed.

1.2.1.1 Statement Label Field

A statement label allows you to reference statements in a program unit. Although any statement can have a label, only labeled executable statements and FORMAT statements can be referred to by other statements. Two statements in a program unit cannot have the same label.

The statement label, which must be a decimal integer, can be positioned in any column, 1 - 5. Leading zeros and blanks are ignored; e.g., 5, 05, and 00005 are the same label.

1.2.1.2 Initial Line

Use column 6 to indicate an initial line or a continuation line. An initial line indicates a single FORTRAN statement. Use a zero or blank (space) in column 6 to indicate an initial line. An initial line cannot be a comment line; however, it can have a statement label. If you do not label it, leave columns 1 through 5 blank.

1.2.1.3 Continuation Line

A continuation line has no statement label. Using the continuation indicator causes the compiler to consider the last character of the previous line (not comment lines) as part of the line where the continuation indicator is placed. The continuation indicator is any character except 0 or blank (space) in column 6.

1.2.1.4 Statement Text Field

The statement text begins in column 7 and continues to the end of the line *or until an exclamation mark is encountered.* (See Appendix D for the maximum allowed line length.) The interpretation of a statement is not affected by spaces and tabs except when they appear within character and Hollerith constants. To continue a statement on the next line, use the continuation indicator.

Two statements, separated by a semicolon (;), may appear on the same line. The character following the semicolon is treated as column 7 of the second statement. Thus, the statement following the semicolon cannot be a comment, specify a continuation field, or contain a label.

1.2.1.5 Debug Statements

CONVEX FORTRAN allows you to place a debugging statement indicator, the letter D, in column 1 of the statement label field. You may add a statement label in the remaining columns of the label field. To continue a debugging statement over more than one line, begin each new continuation line with a D in column 1 and a continuation indicator.

You may treat debugging statements as comments or as source text to the compiler. If you use the compiler command line option -dc, the statements are treated as source text to the compiler; omitting this option causes the statements to be treated as comments.

1.2.1.6 Compiler Directives

A compiler directive provides information to the compiler or instructs the compiler to override certain conditions that inhibit optimization, vectorization, or parallelization. A compiler directive begins with C$DIR in columns 1 through 5 and must fit on one line. Appendix C describes the compiler directives.

1.2.2 ANSI Standard Formatting

The ANSI FORTRAN standard specifies a restricted form of character-per-column formatting and states that all lines are 72 characters in length. *You may get the same effect with CONVEX FORTRAN by specifying the -72 option on the compiler command line. If -72 is specified, any line shorter than 72 characters is padded with blanks and any characters beyond 72 are ignored. A tab counts as one character.*

1.2.3 Tab-Key Formatting

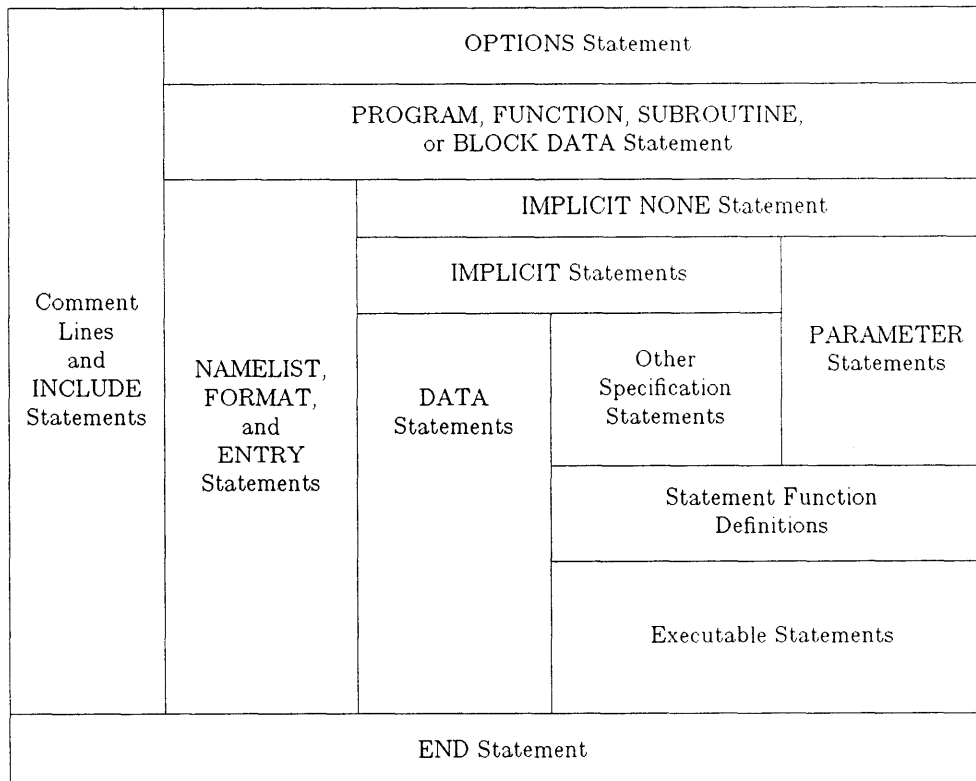
Tab-key formatting is a shorthand method of skipping to the various fields of a CONVEX FORTRAN statement. As with character-per-column formatting, the statement label must appear in the first 5 columns of the line. The next character is the tab character. If the character immediately following the tab is a digit from 1 to 9, this specifies a continuation as if the character had been in column 6 in character-per-column format.

Any other character after the tab is considered the first character of the statement field, as if it had been entered in column 7 in character-per-column format.

1.3 Order of Statements and Lines

Figure 1-1 shows the order required for a CONVEX FORTRAN program unit. You can mix statements separated by vertical lines but not statements separated by horizontal lines. For example, you can intersperse DATA statements with statement function definitions and with executable statements. You cannot, however, mix statement function definitions with executable statements.

Figure 1–1: Required Order of Statements



1.4 OPTIONS Statement

You can use the OPTIONS statement to include options not specified on the FORTRAN command line or to override options that are specified on the command line. The options remain in effect only within the program unit in which they are defined. If used, the OPTIONS statement must be the first statement in a program unit.

The OPTIONS statement has the form:

OPTIONS option [option...]

Table 7-2 summarizes the options that can be specified in the OPTIONS statement.

Table 1-2: OPTIONS Statement

Option	Description
-a1	*Treat non-character arrays as assumed-size arrays.*
-cs	*Check subscript bounds at runtime.*
-F66	*Select FORTRAN-66 rules.*
-i2 (4, or 8)	*Default to integer size.*
-na	*Suppress advisory messages.*
-no	*Perform no optimization.*
-nw	*Suppress warning messages.*
-On	*Perform optimization, where:* *-O0 = local scalar optimization* *-O1 = global scalar optimization* *-O2 = vectorization* *-O3 = automatic parallelization*
-r4 (or 8)	*Default to real size.*
-re	*Generate reentrant code.*
-rl	*Perform loop replication optimizations.*
-sa	*Prevent compiler from generating precompiled argument packets.*
-uo	*Perform potentially unsafe optimizations.*

Example:

OPTIONS -O2 -r8

This statement specifies that the following program unit should be compiled at optimization level -O2 and that all REAL data should be compiled as DOUBLE-PRECISION.

1.5 INCLUDE Statement

The INCLUDE statement causes the compiler to insert source code from the specified file into the program being compiled. The statement has the form:

INCLUDE 'filename'

or

#include 'filename'

where 'filename' is the pathname of the file from which source code is to be read. The contents of the file are inserted at the place where the INCLUDE statement appears.

When the compiler reaches the end of the included file, compilation resumes with the statement following the INCLUDE statement. The included file can itself contain an INCLUDE statement; INCLUDE statements can be nested up to the system limit as described in Appendix D. The INCLUDE statement can appear anywhere within a program unit.

NOTE

The INCLUDE statement operates somewhat differently under COVUEshell. Please refer to the CONVEX COVUEshell Reference Manual for more details.

Chapter 2
Constants, Variables, and Arrays

The basic components of FORTRAN statements are constants, variables, arrays, expressions, and function references.

2.1 Symbolic Names

Variables, arrays, and functions have symbolic names that identify them in a program. A symbolic name must start with a letter and may be followed by any number of letters (A-Z), digits (0-9), *underscores (_), or dollar signs ($), up to the maximum length allowed for a symbolic name (see Appendix D).*

Letters may also be specified in lowercase (a-z) and are converted to uppercase by the compiler; thus the symbolic names ABC and abc are the same. Because dollar signs are used in CONVEX-supplied software, it is recommended that you not use dollar signs in your symbolic names.

2.2 Data Types

Each basic component of a FORTRAN program has a data type. For components with symbolic names, you can specify the data type explicitly in a type statement or implicitly by the first letter of the name. Table 2-1 lists the data types.

Table 2-1: Data Types

Data Type...	Includes Types...
CHARACTER	CHARACTER*n, CHARACTER*(*)
COMPLEX	COMPLEX*8, *COMPLEX*16*
INTEGER	*INTEGER*1 (BYTE), INTEGER*2,* INTEGER*4, *INTEGER*8*
LOGICAL	*LOGICAL*1, LOGICAL*2,* LOGICAL*4, *LOGICAL*8*
REAL	REAL*4, REAL*8 (DOUBLE PRECISION)
RECORD	

In CONVEX FORTRAN, INTEGER*4 corresponds to the standard INTEGER data type, REAL*4 to REAL, COMPLEX*8 to COMPLEX, and REAL*8 to DOUBLE PRECISION. COMPLEX is an ordered pair of real values representing the real and imaginary parts of a complex number. *The COMPLEX*16 data type differs from COMPLEX*8 in that its parts are double-precision rather than single precision. BYTE is a synonym for INTEGER*1.*

The data type determines the storage requirement for the component as shown in Table 2-2. *For LOGICAL and INTEGER, the storage requirement can be controlled by the -i compiler option; the default is four bytes. For REAL and COMPLEX, the storage requirements can be controlled by the -r compiler option.*

For CHARACTER data type, *len* may have an integer value ranging from 1 to the maximum length permitted by the system (see Appendix D). The notation CHARACTER*(*) specifies an assumed-length character string.

Table 2-2: Storage Requirements for Data Types

Data Type	Storage Requirements (Bytes)
LOGICAL	1, 2, 4, or 8
*LOGICAL*1*	*1*
*LOGICAL*2*	*2*
LOGICAL*4	4
*LOGICAL*8*	*8*
INTEGER	1, 2, 4, or 8
*INTEGER*1, BYTE*	*1*
*INTEGER*2*	*2*
INTEGER*4	4
*INTEGER*8*	*8*
REAL	4 or 8
REAL*4	4
REAL*8, DOUBLE PRECISION	8
COMPLEX	8 or 16
COMPLEX*8	8
*COMPLEX*16*	*16*
CHARACTER*len	len
CHARACTER*(*)	

2.2.1 Conversion of Data Types

Where data type differs between the variable or array element on the left side and the expression on the right side of an assignment statement, CONVEX FORTRAN converts the expression on the right to the same data type as that on the left side. For the purpose of conversion, CONVEX FORTRAN does the following:

- *Treats LOGICALS as INTEGERS of the same length.*
- *Converts INTEGERS to longer ones by sign extension and to shorter ones by truncation. Truncation of significant bits causes an integer overflow.*
- Converts INTEGER values to REAL values by truncation. For example, 82762035 is too large to fit in a REAL*4 without loss of precision, so just enough rightmost bits of the binary representation are truncated to make it fit. After conversion, the value becomes 82762033.0
- Converts REAL values to INTEGER values by truncation. Rounding is not performed. For example, I = 5.9 assigns the value 5 to I.
- Converts REAL*8 to REAL*4 by rounding the REAL*8 value to the precision of a REAL*4.

- Converts COMPLEX values to other noncomplex numeric data types by converting the real part only. For example, R = (15.6d0,7.5D6) assigns the value 15.6 to R.
- Converts noncomplex values to COMPLEX by converting first to the appropriate precision REAL value to get the real part, and then assigning 0.0 or 0.0D0 to the imaginary part.
- *Handles COMPLEX-to-COMPLEX conversions by converting the real and imaginary parts separately, that is, as two REAL conversions.*

2.3 Constants

A constant is an arithmetic or logical value, or a character string that does not change during program execution. The form in which a constant is expressed determines the value and data type. The PARAMETER statement is used to assign a symbolic name to a constant.

2.3.1 Integer Constants

An integer constant consists of the digits 0 to 9 and, possibly, a sign. If the constant is positive, the sign is optional; if the constant is negative, the sign is required. Leading zeros have no effect on the value.

Examples:

Legal	Illegal	Reason
248	24.8	Has decimal point
54	5E8	Uses exponential notation
12333	12,000	Has comma

An integer constant is represented at the default integer precision (4) or at a higher precision if the constant is too large for the default precision. You can change the default precision by using compiler options (such as -i or -cft) as described in Appendix B.

2.3.2 Real Constants

A REAL*4 constant consists of an optional positive sign or required negative sign, digits (0-9), a decimal point, and an optional exponent. The exponent is represented as the letter E followed by an integer that denotes the power of 10. You can place the decimal point anywhere in the string (e.g., 2.1, .2, 678912.). When you specify an exponent, the decimal is optional; 7.E3 is the same as 7E3.

A double-precision (REAL*8) constant is identical to the REAL*4 except the exponent is not optional and the exponent letter D is used instead of E.

Examples:

Legal	Illegal	Reason
2500.	2500	Decimal point missing
+2.0E2	2.0E	Exponent field missing
5E4	5,000	Has comma
4E-2		
3.0E4		

2.3.3 Complex Constants

Both COMPLEX*8 and COMPLEX*16 constants consist of a pair of real constants separated by a comma and enclosed in parentheses. The first constant is the real part and the second constant is the imaginary part. A COMPLEX*8 constant is a pair of integer or REAL*4 constants. *A COMPLEX*16 constant is an ordered pair of integer, REAL*4 or REAL*8 constants, where one of the pair must be REAL*8.*

Example:

Legal	*Illegal*	*Reason*
(1.6405D0,-1.6405D0)	*(1.640D)*	*Second constant missing*

2.3.4 Octal Constants

An octal constant consists of one or more octal digits enclosed in apostrophes and followed by the letter O. An octal digit can range from 0 to 7. An octal constant has the form:

'cc...c'O

where c represents an octal digit.

Examples:

Legal	*Illegal*	*Reason*
'765'O	*'835'O*	*8 not in range 0 to 7*
'123'O	*123O*	*Missing apostrophes*

If you specify the -vfc compiler option (see Appendix H), the compiler accepts octal constants of the form:

"cc...c

where c represents an octal digit.

2.3.5 Hexadecimal Constants

A hexadecimal constant consists of one or more hexadecimal digits enclosed in apostrophes and followed by the letter X. A hexadecimal digit can range from 0 to 9, A to F (or a to f). A hexadecimal constant has the form:

'cc...c'X

where c represents a hexadecimal digit.

Examples:

Legal	*Illegal*	*Reason*
'1A6'X	*'FFG'X*	*G not in range 0 - 9, A - F*
'123'X	*'12.4'X*	*Decimal point not allowed*
'FFFFFFFF'X	*1AB2X*	*No apostrophe*
'abc123ff'X		

2.3.5.1 Octal and Hexadecimal Constants—Data Specification

Octal and hexadecimal constants assume data types depending on how they are used. The following conditions apply:

- *When you use octal or hexadecimal constants as actual arguments, no data type is assumed.*
- *When you use either of the constants with a binary operator, the data type of the constant matches the data type of the other operand.*
- *When a specific data type is required, that type is assumed for the constant.*
- *In any other context, the type is INTEGER*4 for the constant.*

When the number of digits required exceeds the length of the constant, the leftmost places are filled with zeros. When the length of the constant exceeds the number of digits required, the excess digits are truncated on the left; an error message results if any of the truncated digits are nonzeros. For example:

```
INTEGER*4 i,j
```

where:

```
i = '12'X                     (same as '00000012'X)
j = '7777ffffffff0076'X       (same as 'ffff0076'X)
```

2.3.6 Hollerith Constants

Hollerith constants are strings of printable ASCII characters, preceded by a character count and the letter H. A Hollerith constant has the form:

nHcc...c

where:

n *specifies the number of the characters in the constant (including spaces and tabs).*

c *is a printable ASCII character.*

The value of n must be an unsigned, positive integer greater than zero.

Example:

Valid	*Invalid*	*Reason*
4HHelp	*0H*	*Must contain one character minimum*

Hollerith constants assume data type according to the context in which they are used:

- *When a specific data type is required, that type is assumed for the constant.*
- *When the constant is used as an actual argument, no data type is assumed.*
- *When the constant is used with a binary operator, the data type of the constant is that of the other operand. Although the data type is that of the other operand, the bit pattern is taken from the Hollerith constant.*

- ***In any other context, the constant assumes an INTEGER*4 data type, unless you specify another one using the -i option.***

2.3.7 Logical Constants

A logical constant represents the value true or false. The form must be .TRUE. or .FALSE. (including the delimiting periods).

2.3.8 Character Constants

A character constant consists of a string of printable ASCII characters with delimiting apostrophes. The value of the character constant includes characters, spaces, and tabs between the delimiting apostrophes. The delimiting apostrophes are not part of the value, but every string must begin and end with them. Within a string, use two consecutive apostrophes ('') to represent an apostrophe.

Examples:

```
'final'
'two''s complement'
'double"quote'
```

A character constant can also be delimited by quotation marks (") instead of apostrophes. In either case, the beginning delimiter must be the same as the ending delimiter. When quotation marks are the delimiters, use two consecutive quotation marks ("") within a string to represent quotation marks.

Examples:

```
"begin"
"two's complement"
"double""quote"
'bad string"     (invalid)
```

2.4 Variables

A variable represents a value that can be changed during program execution by an assignment or READ statement. You can assign an initial value to a variable with a DATA statement or a type statement. A variable is associated with a storage location. Whenever a variable is used, the current value in the storage location is referenced.

A variable type is classified as INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER. The data type describes the data, its storage requirements, and its precision. If there is no explicit type statement, the first character of the symbolic name identifies the data type.

Multiple variables can be associated with the same storage location by means of COMMON statements, EQUIVALENCE statements, or actual arguments and dummy arguments in subprogram references. The COMMON statement allows two or more variables in different program units to share the same storage unit. The EQUIVALENCE statement allows variables in the same program unit to share the same storage unit (see Chapter 3).

2.5 Arrays

An array is a set of adjoining storage locations, with one to seven dimensions, identified by a single symbolic name. For example, a one-dimensional array consists of a single column of figures; a two-dimensional array consists of a table of figures. To reference a specific element in a two-dimensional array, specify both a row and column number. Each array element (individual storage unit) is referenced by the array name and a qualifier called a subscript.

All the values in the array have the same data type and any value assigned is converted to the data type of the array. A DATA statement can be used to define an array element or an entire array before program execution. During execution, an array element is defined with an assignment or input statement; the entire array may be defined with an input statement.

2.5.1 Array Declaration

DIMENSION, COMMON, or type statements allow array declarations. An array declarator defines the name of the array within the program unit, the number of dimensions in the array, and the upper and lower bounds of elements in each dimension. If an array has more than one dimension, separate the dimension declarators with commas.

Examples:

```
DIMENSION MYRAY(5)         ! One-dimensional array with 5 elements; integer type
                           ! by name.

COMMON RERAY(5,5)          ! Two-dimensional array with 25 elements; real type by
                           ! name.

INTEGER A(5,5,5)           ! Three-dimensional array with 125 elements (5 planes,
                           ! 5 rows, 5 columns); INTEGER type statement overrides
                           ! the name rule and any conflicting IMPLICIT statement.

CHARACTER*8 MYRAY(2)       ! One-dimensional character array with 2 elements,
                           ! MYRAY(1) and MYRAY(2); storage space is 8 for each
                           ! element.
```

If you do not specify a lower or upper bound, the lower bound is 1 and the upper bound is the number of elements in that dimension. To use a lower bound that is not 1, you must specify both bounds. The bounds values can be positive, negative, or zero. Separate lower- and upper-bound values by a colon.

The following statement specifies a one-dimensional array with five elements, 0 - 4:

```
DIMENSION MYRAY(0:4)
```

The following statement specifies a two-dimensional array with 25 elements. The first dimension contains elements -1 through 3; the second, elements 2 through 6.

```
COMMON MYRAY(-1:3,2:6)
```

2.5.2 Array Subscripts

A pair of parentheses enclosing one to seven subscript expressions separated by commas constitutes a subscript. A subscript specifies which array element is being referenced. It immediately follows the array name. Specify one subscript expression for each dimension defined for the array. For a two-dimension array of COMMON MYRAY(5,5), a valid reference is MYRAY(2,4). A subscript expression can be any valid arithmetic expression.

2.5.3 Array Storage

Even though array elements are arranged and referenced in dimensions, array storage in memory is in linear sequence. For example, a one-dimensional array consists of a column of figures. In this case, the array is stored with the first element in the first storage location and its last element in the last storage location of the sequence. Multidimensional array elements are stored so the value of the first subscript (leftmost) varies most rapidly. This is called the "order of subscript progression."

2.6 Expressions

An expression is a combination of one or more operands and optional operators. During program execution, an expression specifies a computation or evaluation that produces a single value. The operators determine the operations to be executed on the operands. An expression can be arithmetic, logical, relational, or character.

2.6.1 Arithmetic Expressions

An arithmetic expression consists of arithmetic operators and operands. Arithmetic operands include character, ***Hollerith, octal, and hexadecimal*** constants described previously, in addition to the standard FORTRAN-77 operands of numeric constants, numeric variables, numeric array elements, arithmetic expressions enclosed in parentheses, or arithmetic function references. ***The term "numeric" as used here includes logical data. The system treats logical data as integer data when it is used in an arithmetic context.***

The arithmetic operator specifies the type of computation to perform on the operands. This computation generates a numeric value. The arithmetic operators are:

Operator	Function	Example
**	Exponentiation	C**2
*	Multiplication	C*2
/	Division	C/2
+	Addition	C+2
-	Subtraction	C-2
+	Unary plus (identity)	C/(+2)
-	Unary minus (negation)	C/(-2)

Arithmetic operators used with two operands are binary operators. The plus and minus operators are unary, however; they change or confirm the sign of the operand.

2.6.1.1 Operator Precedence

When an expression contains two or more operators, they are executed according to the following order of precedence if there are no parentheses.

Operator	Priority
**	Evaluated first
* and /	Evaluated second
+ and -	Evaluated last

Operators with the highest priority are processed before those with lower priority except that parentheses within an expression cause the operation inside the parentheses to be performed first.

Examples:

```
6 * 2**2 - 5            ! Yields a value of 19
3 + 4 * 3 - 9/3         ! Yields a value of 12
(3 + 4) * 3 - 9/3       ! Yields a value of 18
```

When an expression contains two or more operators of equal precedence, evaluation occurs in left-to-right order, except for exponentiation that is evaluated right to left. CONVEX FORTRAN, however, may execute operations in differing orders as long as the order remains algebraically equivalent to left-to-right order of evaluation.

If more than one set of operators appears within parentheses, the operators are evaluated according to the normal order of precedence, unless overridden by parentheses within parentheses. In nested parenthesized expressions, the innermost set is evaluated first.

2.6.1.2 Data Type Priority

Where operands of different data types are combined in an arithmetic expression, the higher-ranked argument determines the type, and the greater-precision argument determines the precision. Data types are ranked as follows:

Rank	Data Type
1 (lowest)	*LOGICAL*1, *2, *4, *8*
2	*INTEGER*1 (BYTE), *2,* *4, *8*
3	REAL*4 (REAL), *8 (DOUBLE PRECISION)
4 (highest)	COMPLEX*8 (COMPLEX), **16 (DOUBLE COMPLEX)*

*When LOGICAL and INTEGER are combined, the resulting type is INTEGER, and the precision is the highest specified, whether *1, *2, *4, or *8. Even REAL*8 and COMPLEX*8 yielding COMPLEX*16 is consistent with this rule, if you regard the precision of COMPLEX*8 as 4 and of COMPLEX*16 as 8 (the precision of the real and imaginary parts). The CONVEX FORTRAN extensions in data types are LOGICAL*1, LOGICAL*2, LOGICAL*8, INTEGER*1, INTEGER*2, INTEGER*8, and COMPLEX*16.*

The data types of arithmetic expressions follow certain conventions:

- *LOGICAL entities are treated as INTEGERS when used in an arithmetic context.*
- *REAL operations are performed only if one or more of the operands is REAL.*
- *In COMPLEX operations, those operations involving COMPLEX*8 and REAL*8 elements are evaluated as COMPLEX*16 operations; hence, the REAL*8 element is not rounded.*

*These conventions also apply to arithmetic operations where one of the operands is a constant. Additional precision is used for the constant if a real or complex constant is used in a higher-precision expression. In such expressions, the effect is as if a REAL*8 representation of the constant had been given.*

2.6.2 Relational Expressions

A relational expression compares either the value of two arithmetic expressions or the value of two character expressions that produce a logical value of true or false. The two expressions are separated by a relational operator. Each relational operator must include delimiting periods. The relational operators are .LT., .LE., .EQ., .NE., .GT., and .GE.

2.6.3 Logical Expressions

A logical expression consists of one logical operand or a combination of logical operands and logical operators. After evaluation, a logical expression produces a logical value of true or false. Logical operands in CONVEX FORTRAN can be any of the following:

- A logical constant *or an integer.*
- A logical variable *or an integer.*
- A logical array element *or an integer.*
- A logical expression enclosed in parentheses *or an integer.*
- A logical function reference *or an integer.*
- A relational expression.

The evaluation of a logical expression that contains two or more logical operators is based on operator precedence as follows:

Precedence	Operator
Lowest	.EQV., .NEQV. *(.XOR.)*
	.OR.
	.AND.
Highest	.NOT.

The logical operator .XOR. is the same as .NEQV. Operators on the same level of precedence are interpreted from left to right. The arithmetic rules for operator precedence apply for evaluation. The evaluation of mixed operations is first by arithmetic rules of precedence, next by relational operations, and last by logical operations.

The expression in the following example yields a value of FALSE:

```
K = 6
M = 2
N = 5
(K .LE. N) .AND. (N .GT. M)
```

As stated in the ANSI standard, logical operators used on logical values produce values of type LOGICAL. *Logical operators operating on integer values produce values of type INTEGER. The logical operation is carried out bit by bit on the corresponding bits of the internal binary representation of the integer elements. When a logical operator combines integer and logical values, the logical value is first converted to an integer. The operation is then carried out for the two integer elements; the resulting data type is INTEGER.*

2.6.4 Character Expressions

The evaluation of a character expression produces a string of type CHARACTER. Within a character expression, two slashes may be used to specify concatenation. Concatenation produces a string that is the sum of the operand strings and executes from left to right.

Parentheses have no effect on the value of a character expression. If spaces are included in the expression, the spaces are part of the value.

Examples:

```
'MY'//'EXAMPLE'      ! Yields a value of MYEXAMPLE
'MY '//'EXAMPLE'     ! Yields a value of MY EXAMPLE
```

2.6.4.1 Character Substrings

A character substring is a sequence of adjacent characters that are part of a character variable or array element. A substring name identifies a character substring that can be assigned values and referenced. A character substring reference to a variable has the form:

var ([*expr1*] : [*expr2*])

where *var* is a character variable name, *expr1* is an optional numeric expression indicating the leftmost character position of the substring, and *expr2* is an optional numeric expression indicating the rightmost character position of the substring. If *expr1* is omitted, a value of 1 is assumed; if *expr2* is omitted, the length of the character variable is assumed. The value of *expr1* must be positive and less than or equal to *expr1*. The value of *expr2* must be less than or equal to the length of the string.

For example, if:

```
CHARACTER*14 NAME
NAME = 'CONVEX FORTRAN'
```

then:

```
NAME(8:14)
```

indicates a substring beginning with the position 8 (F) and ending in position 14 (N) of the variable NAME, giving the value of FORTRAN to the substring NAME(8:14). Character positions are numbered from left to right within a character variable or array element.

The character substring form for referencing an array is: character array name, subscript expression, numeric expression indicating leftmost character of substring, a colon (:), and numeric expression indicating rightmost character of substring. For example:

```
EXAMPLE(1,5)(:3)
```

indicates the substring begins with the first character position and ends with the third character position of the character array EXAMPLE(1,5).

2.6.4.2 Constant Expressions

A constant expression is one in which each primary is a constant, the symbolic name of a constant, or a constant expression.

A compile-time constant expression can be a compile-time logical expression, character expression, or arithmetic expression. CONVEX FORTRAN provides the following extensions:

- In the compile-time logical expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or ***one of the functions IAND, IOR, NOT, IEOR, ISHFT, LGE, LGT, LLE, or LLT with constant operands.***

- In the compile-time character expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or ***the function CHAR with a constant operand.***

- In the compile-time arithmetic expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or ***one of the functions MIN, MAX, ABS, MOD, ICHAR, NINT, DIM, DPROD, CMPLX, CONJG, or IMAG with constant operands.***

Chapter 3
Specification Statements

Specification statements are nonexecutable statements that appear before the first executable statement in a program unit. These statements define the type of variable or array, stipulate storage requirements for each variable based on its type, indicate the dimension of arrays, define storage sharing, and assign initial values to variables and arrays. Specification statements include:

- COMMON
- IMPLICIT
- Type declaration
- DIMENSION
- EQUIVALENCE
- PARAMETER
- PROGRAM
- *NAMELIST*
- EXTERNAL
- INTRINSIC
- SAVE

If you specify the -sfc (Sun FORTRAN) compiler option, you may use the STATIC and AUTOMATIC statements. For further information, see Appendix I. The DATA statement, which assigns initial values to variables, arrays, and array elements, is classified as an initialization statement. The DATA statement is described in Chapter 4.

3.1 COMMON Statement

The COMMON statement allows variables or arrays in a main program or subprogram to share the same storage location with variables and arrays in other subprograms. These blocks of storage are called common blocks. Common blocks can be named or unnamed; unnamed blocks are called blank common. The block specification determines storage order of variables and arrays. Named common blocks of the same name may be of different sizes in different program units of an executable program.

The COMMON statement has the form:

COMMON [/[*cbn*]/] *nlist* [[,] /[*cbn*] /*nlist*]...

where:

cbn is a symbolic name for a common block. If you do not specify a symbolic name (blank common), the first pair of slashes is optional.

nlist is a list of variable names, array names, and array declarators.

An entity name (name in *nlist*) can appear only once in a COMMON statement within a program unit. If a common block name appears twice in the same program unit, the effect is as if the *nlist* of the second appearance came after the first *nlist*. You can use a common block name (*cbn*) more than once in a COMMON statement and in a program unit. A common block can have the same name as any local entity except a constant, intrinsic function, or a variable name that is also a function name. If you give both a common block and variable the same name, all references to the name, except when it appears surrounded by slashes (/) in COMMON and

SAVE statements, indicate the variable. Thus, SAVE X refers to the variable, while SAVE /X/ refers to the common block.

Arrays and variables in COMMON can be initialized in DATA statements in program units other than BLOCK data subprograms. A variable in COMMON, however, can be initialized only in one program unit, although different variables in a COMMON block may be initialized in different program units.

3.2 IMPLICIT Statement

The IMPLICIT statement allows you to override the implied data typing of symbolic names within a program unit. The IMPLICIT statement has the forms:

> **IMPLICIT** *typ* (*a*[,*a*]...) [,[*typ*] (*a*[,*a*]...)]...
> *IMPLICIT NONE*

where:

typ is INTEGER*[*len]*, REAL*[*len]*, DOUBLE PRECISION, LDOUBLE COMPLEX, COMPLEX*[*len]*, LOGICAL*[*len]*, or CHARACTER [**len*] data type.

a is one letter or a range of letters; the range is expressed as first letter of range, minus sign, last letter of range, i.e., F-H.

len is an optional length specifier for the data type.

If used, the IMPLICIT statement must precede any other specification statements in the program unit. If this statement is not used, variable names that begin with the letters I through N imply type integer; all others imply REAL types.

The IMPLICIT NONE form of the statement overrides all implicit defaults. In this case, you must explicitly declare the data types of all symbolic names in the program unit. If you specify IMPLICIT NONE, no other IMPLICIT statement can be included in the program unit.

Examples:

```
IMPLICIT COMPLEX(F,H-J)     ! Any name beginning with the letter F or any of
                              the letters H, I, J is type COMPLEX

IMPLICIT LOGICAL(L)         ! Any name beginning with the letter L is type
                              LOGICAL (value of .TRUE. or .FALSE.)

IMPLICIT CHARACTER*8(C)     ! Any name beginning with the letter C is type
                              CHARACTER with the length of the character entity
                              being 8

IMPLICIT REAL (A-H), (O-Z) ! Any name beginning with the letters A through H
                              or O through Z is type REAL
```

3.3 Type-Declaration Statements

Type statements are numeric or CHARACTER type declarations. Type statements take precedence over the name rule and IMPLICIT statements if there is a conflict. You can also use these statements to specify array dimensions.

Both numeric and CHARACTER type-declaration statements can initialize data by including values bounded by slashes (/) in the statement. Place the values after the symbolic name of the variable or array to be initialized. Initial values are assigned in the same way that they are assigned in DATA statements. The following subsections include examples of both forms of type-declaration statements.

3.3.0.1 Numeric Type-Declaration Statements

As mentioned previously, type-declaration statements have the form:

type v [/clist/] [,v [/clist/]]...

where:

type is any data type specifier except CHARACTER.

v is the symbolic name of a constant, variable, array, statement function or function subprogram, or array declarator.

clist *is a list of constants. (See "Data Statement," Chapter 4.)*

*You can follow the symbolic name with a data-type length specifier written as *s, where s is one of the acceptable lengths for the data type being declared. A new length, which overrides the length attribute that the statement implies, is thus assigned to the specified item. When you use data type-length specifiers with an array declarator, place them immediately after the array name.*

You can assign initial values to variables or arrays with /clist/, which initializes the variable or array immediately preceding it. For arrays only, the clist can consist of more than one element. If you initialize an array using /clist/, every element in the array must be assigned a value, as in the following example:

```
REAL*8 PI/7.43562D0/, E/3.33D0/, QARRAY(10)/5*0.0,5*1.0/
```

3.3.0.2 CHARACTER Type-Declaration Statements

CHARACTER type-declaration statements, like the numeric type, use the */clist/* provision but in the following form:

CHARACTER [**len* [,]] *v*[**len*] */[clist/]* [,*v*[**len*] */[clist/]*]...

where:

v is the symbolic name of a constant, variable, array, statement function or function subprogram, or array declarator.

len is an unsigned integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The value of *len* specifies the length of the character data elements. When an array is being declared, the length must appear after the array dimension.

clist *is a list of constants, as in the DATA statement.*

As for numeric type-declaration statements, the /clist/ assigns initial values to the variable or the array immediately preceding it. For arrays only, clist can contain more than one element. Where this is the case, it must contain a value for every element in the array.

The following example specifies an array DEMO consisting of fifty 16-character elements, an array DUMMY comprising twenty 9-character elements, and a variable DRAFT, which is 5 characters long ***with an initial value of 'ABCDE':***

```
CHARACTER*16 DEMO(50), DUMMY(20)*9, DRAFT*5 /'ABCDE'/
```

If you do not specify the length of an item (CHARACTER**len*), its length is *len*—the default length specification for that item. If you specify the length, that length takes precedence over the default length specified in CHARACTER**len*.

If you specify the length as an *, i.e., CHARACTER *(*), the function name or dummy argument assumes the length specification of the corresponding function reference or actual argument. This is known as an assumed-length character argument.

3.4 DIMENSION Statement

The DIMENSION statement names arrays and specifies the bounds of the array. The type of array and the product of the subscripts determine the number of storage units allocated to each array named in the statement. The name rule or a preceding IMPLICIT or type statement determines the data type. The general form includes the array name and the array dimension. Each dimension is defined by a dimension declarator within the array declarator.

Example:

```
DIMENSION MYRAY(4,5)
```

The preceding example specifies a type INTEGER two-dimensional array with 20 elements. The product of the dimension declarators, (4,5), determines the total number of storage elements assigned to the array. When you specify two or more array declarators, separate them by commas.

3.5 EQUIVALENCE Statement

The EQUIVALENCE statement causes two or more entities within a program unit to refer to the same storage area. Thus, the same storage unit can be referenced by more than one name. Each statement consists of two or more variables, array elements, array names, or substring names separated by a comma. An array must be dimensioned with a DIMENSION, type, or COMMON statement before it or any of its elements can be equivalenced. All elements contained in the same set of parentheses are allotted storage in the same location. For example:

```
DIMENSION CONVEX (10), COM (12)
EQUIVALENCE (F,G,H), (CONVEX(9), COM(10)), (L,M,N)
```

indicates that the variables F, G, and H share the same location; the 9th element in array CONVEX and the 10th element in array COM share the same location; the variables L, M, and N share the same location.

If different data types are equivalenced, the EQUIVALENCE statement does not imply mathematical equivalence or type conversion. Type is associated with the name used to reference a location; the name determines how data is stored or read from the location. Names of dummy arguments of an external procedure in a subprogram, or a variable name that is a function name cannot appear in an EQUIVALENCE statement.

Use of the EQUIVALENCE statement can interfere with optimization of the program.

3.5.0.3 Equivalencing Arrays

Making one array element equivalent to an element of another array also defines the relative locations of the other array elements. For example:

```
DIMENSION A(5), B(4,3)
EQUIVALENCE (A(2),B(2,2))
```

The entire array A shares part of the storage space allotted to array B. The EQUIVALENCE statements:

```
EQUIVALENCE (A,B(1,2)) or EQUIVALENCE (A(5),B(1,3))
```

also align the two arrays in the same manner as EQUIVALENCE (A(2),B(2,2)). Table 3-1 shows how these statements align the arrays.

Table 3-1: Array Locations

Array B		Array A	
Elements	Location Number	Elements	Location Number
B(1,1)	1		
B(2,1)	2		
B(3,1)	3		
B(4,1)	4		
B(1,2)	5	A(1)	1
B(2,2)	6	A(2)	2
B(3,2)	7	A(3)	3
B(4,2)	8	A(4)	4
B(1,3)	9	A(5)	5
B(2,3)	10		
B(3,3)	11		
B(4,3)	12		

The same storage location cannot be allotted to two or more elements of the same array. For instance, the following statements are invalid because they allocate the same storage for A(1) and A(3).

```
DIMENSION A(5), B(4,3)
EQUIVALENCE (A(1),B(3,3)), (A(3),B(3,3))
```

When making arrays equivalent for storage, you can identify an array element with a single subscript (the linear element number), even if the array was defined as a multidimensional array. For example:

```
REAL A(10,10),Z(200)
EQUIVALENCE (A(100),Z(150))
```

associates element A(10,10) with element Z(150).

3.5.0.4 Equivalencing Substrings

When you make character substrings equivalent for storage, the EQUIVALENCE statement also defines the storage location for the other corresponding characters in the character entities.

For example:

```
CHARACTER PROD*12, N*8
EQUIVALENCE (PROD(6:10), N(4:8))
```

specifies the character variables PROD(6) and N(4), PROD(7) and N(5), through PROD(10) and N(8) share storage locations. Thus, the storage locations for the other character entities are defined as: N(1) shares storage with PROD(3), N(2) with PROD(4), and so forth.

Assigning the same storage location to two or more character substrings that begin at different character positions within the same character variable or array is prohibited. You cannot use EQUIVALENCE statements to indicate that consecutive storage units are to be nonconsecutive.

3.5.0.5 Using EQUIVALENCE in Common Blocks

You can extend a common block of storage with the EQUIVALENCE statement if you extend locations beyond the last element and do not add to the beginning of the common block.

For example:

```
DIMENSION A(5), B(2,3)
COMMON B
EQUIVALENCE (A(1), B(1,2))
```

extends the common block beyond the last element. The existing common block includes B(1,1) through B(2,3) and A(1) through A(4); A(5) is the extended portion being added beyond the last element B(2,3) of the existing common block. If you change COMMON B in the previous example to COMMON A:

```
DIMENSION A(5), B(2,3)
COMMON A
EQUIVALENCE (A(1), B(1,2))
```

the extension is invalid. The common block now includes A(1) through A(4), and B(1,2) through B(2,3); both B(1,1) and B(2,1) comprise the extended portion preceding the common block, which is invalid.

3.6 PARAMETER Statement

CONVEX FORTRAN supports two types of PARAMETER statements. The first type is the standard FORTRAN PARAMETER statement; the second type is for compatibility with compilers supplied by other vendors.

3.6.1 Standard PARAMETER statement

The PARAMETER statement assigns a symbolic name to a constant to be used within the program unit. The name specified is used to reference that constant in other statements in the program unit. You must have previously defined any symbolic constant names that appear in an expression. A constant is named only once in a program although you can use the symbolic name in subsequent DATA statements or expressions in the same program.

You can use the PARAMETER statement to define an entire format specification; however, it must not appear as a part of a format specification. Also, it cannot be used as part of another constant although *CONVEX FORTRAN provides for use as either the real or imaginary part of a complex constant.*

Examples:

```
PARAMETER (SMITH = 1)
PARAMETER (BOSQUE = 10, HAYS = 100)
COMPLEX LAMAR
PARAMETER (LAMAR = (BOSQUE,HAYS))
```

To determine the data type associated with each constant, use the implicit naming convention or an explicit type-declaration statement appearing before the PARAMETER statement. If the length for a character-type constant is greater than the default length of 1, you must specify the length before the first appearance of the symbolic name.

3.6.2 Alternate PARAMETER Statement

An alternate form of the PARAMETER statement is available. This PARAMETER statement also assigns a symbolic name to a constant but its list is not bounded by parentheses and the form of the constant determines the data type of the variable. The alternate PARAMETER statement does not conform to the ANSI standard but has the following form:

PARAMETER p=c [, p=c]...

where p is a symbolic name and c is a constant, the symbolic name of a constant, or a compile-time constant expression.

The data type is not determined by the implicit or explicit typing of the symbolic name, but by the form of the constant. Once you have defined a symbolic name as a constant, you can use it wherever a constant is allowed. You cannot, however, use the symbolic name of a constant as part of another constant, but you can use it as a real or imaginary part of a complex constant.

Since the symbolic name of a constant assumes the data type of its corresponding expression, you cannot specify the data type of a parameter in a type declaration statement, nor does the initial letter of the constant name affect the data type.

Example:

```
PARAMETER BAKER = 3, XRAY = 45.4, ALPHA = XRAY*BAKER
```

To use the alternate PARAMETER statement with only one constant, specify the -vfc compiler option. (See Appendix H.)

3.7 PROGRAM Statement

The PROGRAM statement can be used to assign a name to the main program unit; its use is optional. If used, a PROGRAM statement must always be the first statement in the program *unless an OPTIONS statement is also included, in which case the PROGRAM statement immediately follows the OPTIONS statement.*

The PROGRAM statement has the form:

PROGRAM *pgm*

where *pgm* is the symbolic name for the main program.

Do not use the symbolic main program name as a name for an external procedure, block data subprogram, or common block in the same executable program. Also, do not use a symbolic main program name as a local name in the main program.

You cannot reference the main program from itself or from a subprogram. An executable program has only one main program.

3.8 NAMELIST Statement

The NAMELIST statement associates a single unique name with a list of variables or array names. This name defines a list of entities that can be modified (read) or transferred (written). Thus, you can use this unique name in namelist-directed I/O statements in place of the entities list. The NAMELIST statement has the form:

NAMELIST /nlgrpname/varlist [[,]/nlgrpname/varlist]...

where:

nlgrpname	*is a symbolic name representing the list of entities to be read or written.*
varlist	*is a list of variable or array names (separated by commas) to be associated with the nlgrpname. A variable or array name may occur in more than one varlist. An entity may be a dummy argument. These entities, which may be explicitly or implicitly typed, can be any data type.*

An entity can be of integer, real, logical, complex, and character data types. If the entity and the constant value assigned to it are not of the same type, the compiler performs the arithmetic assignment conversion. You may not, however, convert between numeric and character data types.

The following example shows the format for namelist-directed input. The $ in the first and last lines of the example are in column two with a space in column 1. The remaining lines begin with a tab.

```
(space)$CONTROL
(tab)TESTCASE='40004.00',
(tab)CONDITION=.FALSE.,
(tab)BEGIN=100,
(tab)REPEAT=10,
(space)$END
```

The following statement illustrates the use of NAMELIST:

```
NAMELIST /EXAM1/ TESTA, TESTB, TESTC /EXAM2/ TOTTEST
```

In this example, the NAMELIST statement defines two groupnames—EXAM1 and EXAM2. The first represents three entities (TESTA, TESTB, and TESTC), while the second represents one entity (TOTTEST). The order in which you list the entities in the varlist determines the order in which the values are output; however, the order of input values is immaterial. Also, you do not need to define every entity in the corresponding varlist during input. For instance, using the previous example, you could input values for only TESTA and TESTB. The value of TESTC would remain unchanged.

Although you cannot use array elements and character substrings in a namelist, you can use namelist-directed I/O to assign values to elements of arrays or substrings of character variables that occur in the namelist. You can also use a variable or an array name in several namelists. (See Chapters 7 and 8 for more information on namelist-directed I/O.)

3.9 EXTERNAL Statement

An EXTERNAL statement identifies a symbolic name as representing an externally defined procedure or dummy procedure. It indicates that a given name is the name of a subprogram instead of a variable or array name. An EXTERNAL statement must be used for a subprogram or dummy procedure name that appears as an actual argument in a function reference or in a CALL statement. The form is:

EXTERNAL *n* [,*n*]...

where *n* is the symbolic name of a user-supplied subprogram, block data subprogram, or dummy procedure.

If an EXTERNAL statement declares an intrinsic name as an external procedure, the intrinsic function cannot be referenced in that program unit. That name is associated with an external procedure. For example, if you declare COS in the EXTERNAL statement (EXTERNAL COS), all subsequent references are to an external procedure COS, not the intrinsic function COS.

3.10 INTRINSIC Statement

The INTRINSIC statement permits a name that specifies an intrinsic function to be used as an actual argument. The INTRINSIC statement has the following format:

INTRINSIC *n* [,*n*]...

where *n* is one of the FORTRAN intrinsic functions.

If the name of an intrinsic function is to be used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. Not all types of intrinsic functions can be used as actual arguments. These include intrinsic functions for type conversion, maximum/minimum value and lexical comparison functions, e.g., INT, IFIX, IDINT, REAL, FLOAT, SNGL, DBLE, CMPLX, ICHAR, CHAR, LGE, LGT, LLE, LLT, MAX, MAX0, AMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, and MIN1.

3.11 SAVE Statement

The SAVE statement retains the values of designated variables and arrays in a subroutine or function when a RETURN or END statement is executed. Thus, items specified in a SAVE statement do not become undefined when the subroutine or function completes execution. In the next call to the subroutine or function, a saved item has the same value it had on return from the preceding call.

The SAVE statement has the following format:

SAVE [*n* [,*n*]...]

where *n* is a variable name, array name, or a named common block contained between slashes (e.g., /NCOM/). A dummy argument name, subprogram or function name, or name of an entity within a common block is not allowed. When a common block name appears in a SAVE statement, all the variables and arrays in the common block are saved.

If the SAVE statement contains no arguments, the values of all allowable entities are retained.

Chapter 4
DATA Statement

The DATA statement establishes initial values for arrays, array elements, substrings, and variables. Values are initialized when the program unit is compiled and may be changed during program execution.

The DATA statement is nonexecutable. All entities initialized with a DATA statement are defined when program execution begins; all entities not initialized with a DATA statement are undefined when program execution begins. Undefined entities must be defined before they can be referenced.

4.1 DATA Statement Form

The DATA statement has the following form:

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

where:

nlist is a list of one or more array names, array element names, character substring names, implied-DO lists or variable names. You cannot use dummy argument names or function names in the *nlist*.

clist is a list of constants (numeric, character, logical, or ***Hollerith***) or symbolic names of constants (defined with a PARAMETER statement). Items in *clist* are consecutively assigned to the entities in *nlist*; the first item in *nlist* corresponds to the first item in *clist*. Constants can be repeatedly associated with entities in the *nlist*.

The number of names in the *nlist* must equal the number of constants in the *clist*. The following statement is invalid because there are two values for one variable.

```
DATA MYVAR/5,9/
```

You can repeat the same value for more than one element by placing a nonzero, unsigned integer constant indicating the number of repetitions and an asterisk (*) before the value. For example, the following statement initializes a character value of NAME for C, logical value .FALSE. for LOW, and 3 for both MYEXAMPLE(2) and MYEXAMPLE(3).

```
DATA C,LOW,MYEXAMPLE(2),MYEXAMPLE(3)/'NAME',.FALSE.,2*3/
```

As long as you retain the correct name/value association, the order and grouping is immaterial. The previous example could also be represented as:

```
DATA C/'NAME'/,LOW/.FALSE./,MYEXAMPLE(2),MYEXAMPLE(3)/2*3/
```

When a character entity is longer than its corresponding character constant, blanks are added on the right. If shorter, extra characters on the right are ignored. For example, the following statements initialize PROD to PRODUCT^IS and NAME to GOOD^^^^^^:

```
CHARACTER*10 PROD,NAME
DATA PROD,NAME/'PRODUCT^IS','GOOD'/
```

The character entity is the same length as the constant in PROD, so no blanks are added or ignored. Six blanks are automatically added to GOOD, however, as the character entity is longer than its corresponding character constant.

You can use a DATA statement to initialize all or part of an array. For example:

```
DIMENSION MYRAY(4,3)
DATA MYRAY /12*5/
```

indicates that the value of all MYRAY elements are initialized to 5. Elements of the array are initialized in the order of subscript progression.

4.1.1 Implied-DO

Implied-DO lists can occur in DATA statements in the form of:

(*dlist, i=m1,m2,m3*)

where:

dlist is a list of array element names and implied-DO lists.

i is the name of an integer variable termed the implied-DO variable.

m1,m2,m3 are integer constant expressions which can contain implied-DO variables of other implied-DO lists. The constants specify the initial value, terminal value and increment, respectively, for the integer variable. If you omit the comma and the value of *m3*, the incrementation value defaults to 1. The incrementation count must be positive.

Examples Using Implied-DO:

The statements:

```
REAL C(8),D(12)
DATA E,(C(I),I=2,6,2),F,(D(J),J=1,3)/4*0,4*1/
```

initialize E, C(2), C(4), C(6) to 0.0 and F, D(1), D(2), D(3) to 1.0.

The statements:

```
DIMENSION B(10,10)
DATA ( (B(I,J),I=1,5),J=1,5)/25*2/
```

initialize the 25 elements of the submatrix to 2.0; the matrix is located at the upper-left corner of B.

The following statements initialize 10 elements of the matrix B to 5:

```
INTEGER B(4,4)
DATA ( (B(I,J),J=1,I),I=1,4)/10*5/
```

The matrix contains elements B(1,1...1,4), B(2,2...2,4), B(3,3), B(3,4) and B(4,4).

4.2 DATA Statement Extensions

If, in the DATA statement, both the constant value in clist and the entity in nlist have numeric data types, you can determine the data-type conversion in addition to the standard as follows:

- *If either an octal or hexadecimal constant is assigned to a variable or array element, the data type of the variable or array element determines the number of digits that can be assigned. Where the constant contains fewer digits than the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left.*

- *If either a Hollerith or character constant is assigned to a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the variable or array element. Where the Hollerith or character constant contains fewer characters than the variables or array element, spaces are added on the right of the constant. If the constant contains more characters than can be stored, excess characters on the right of the constant are eliminated.*

The constant value in clist may be of the numeric data type and the entity in nlist of the CHARACTER data type. When this is the case, the entity must conform to the following:

- *The character entity must have a length of one character.*

- *The constant must be an integer, octal, or hexadecimal constant and must have a value in the range 0 through 255.*

*Following these restrictions permits the entity to be initialized with the character that has the ASCII code specified by the constant, which, in turn, allows a character entity to be initialized to any 8-bit ASCII code. The next example initializes the real array R to all zeros, the real variable P, the character variable C*4 to 'TEST', and the character variable CR*1 to the ASCII character code '0D'X.*

```
DATA R(20), PI /20*0.0, 3.14159265/
DATA C, CR /7HTESTING, 'D'X
```

Arrays and variables in COMMON can be initialized in DATA statements in program units other than BLOCK data subprograms. A variable in COMMON can only be initialized in one program unit.

Chapter 5
Assignment Statements

An assignment statement evaluates an expression and assigns the value to a variable, a substring, or an array element. The ASSIGN statement assigns a statement label value to a variable. The ASSIGN statement is discussed later in this chapter. The assignment statement has the form:

$v = ex$

where v is a variable, array element, or substring, and ex is an expression.

If the variable type on the left of the equal sign is the same as that of the expression on the right, the value is assigned directly. For example, in the statement:

```
I = (L + M)/K          L = 9, M = 6, K = 3
```

the variable and the expression are both INTEGER, so the value 5 is assigned directly. If the variable is INTEGER and the expression is REAL, the expression is converted to INTEGER and assigned to the variable. Thus, the statement:

```
I = (R + S)/T          R = 8.0, S = 9.0, T = 3.0
```

assigns the INTEGER value of 5 (17/3 truncated) to I. In this example, I is truncated and converted to INTEGER. In general, if the data types differ, the value of the expression is converted to the type of the entity before the value is assigned.

5.1 Character Conversions

The character assignment statement has the form:

$v = ce$

where v is a character variable, array element, or substring, and ce is a character expression.

The assigned entity and the expression can have different lengths. If the entity (v) is of greater length than the length of the character expression (ce), CONVEX FORTRAN inserts blanks after the characters until the length is equal to v. If the length of v is less than the length of ce, extra expression characters on the right are truncated until v and ce are equal in length. For example, the following statements assign CONVEX^^^^^^ to NAME and *supercompute* (12 characters only) to PROD.

```
CHARACTER*12 NAME, PROD
NAME = 'CONVEX'
PROD = 'supercomputer'
```

The same character positions defined in v cannot be referenced in ce within the same statement. When you assign a value to a character substring, the character positions in the character variable or array element not included in the substring are not affected. If a value was previously assigned, the value remains the same; if the value was undefined, it remains undefined. Using a differing substring with the same name, such as A1(1:3) = A1(4:6), is acceptable.

Table 5-1 summarizes the data conversion rules for assignment statements.

Table 5-1: Conversion of Expressions

Type of Variable (V)	Type of Expression (E)	Value Assigned
INTEGER/LOGICAL	INTEGER/LOGICAL	Direct assignment of E to V.
	REAL	Truncate E to INTEGER and assign to V.
	REAL*8	Truncate E to INTEGER and assign to V.
	COMPLEX	Truncate real part of E to INTEGER and assign to V. Imaginary part not used.
	*COMPLEX*16*	*Truncate real part of E to INTEGER and assign to V. Imaginary part not used.*
REAL	INTEGER/LOGICAL	Convert to REAL and assign to V.
	REAL	Direct assignment of E to V.
	REAL*8	Assign significant digits of E to V; least significant digits of E rounded.
	COMPLEX	Assign real part of E to V. Imaginary part not used.
	*COMPLEX*16*	*Assign significant digits of real part of E to V; least significant digits rounded. Imaginary part not used.*
*REAL*8* (DOUBLE PRECISION)	INTEGER/LOGICAL	Convert to REAL and assign to V.
	REAL	Assign E to most significant portion of V. Least significant portion of V is assigned 0.
	REAL*8	Direct assignment of E to V.
	COMPLEX	Assign real part of E to most significant of V; assign 0 to least significant of V. Imaginary part not used.
	*COMPLEX*16*	*Assign real part of E to V. Imaginary part not used.*
COMPLEX	INTEGER/LOGICAL	Convert to REAL and assign to real part of V. Assign 0.0 to imaginary part of V.
	REAL	Assign E to real part of V. Assign 0.0 to imaginary part of V.
	REAL*8	Assign most significant digits of E to real part of V; least significant portion of E is rounded. Assign 0.0 to imaginary part of V.
	COMPLEX	Direct assignment of E to V.
	*COMPLEX*16*	*Assign most significant portion of E to real part of V; least significant portion of real part of E is rounded. Assign most significant imaginary part of E to imaginary part of V; least significant portion of imaginary E is rounded.*

Table 5-1: Conversion of Expressions (continued)

Type of Variable (V)	Type of Expression (E)	Value Assigned
COMPLEX*16	*INTEGER/LOGICAL*	*Convert to REAL and assign to V. Assign 0.0 to imaginary part of V.*
	REAL	*Assign E to most significant portion of real part of V. Assign 0.0 to imaginary part of V.*
	*REAL*8*	*Assign E to real part of V. Assign 0.0 to imaginary part of V.*
	COMPLEX	*Assign real part of E to most significant of real V; assign 0 to least significant portion of real part. Assign imaginary part of E to most significant of imaginary V; assign 0 to least significant portion of imaginary part.*
	*COMPLEX*16*	*Direct assignment of E to V.*
RECORD	*RECORD*	*Must be a RECORD of the same type.*

5.2 ASSIGN Statement

To assign a statement label to an integer variable, use the ASSIGN statement. It has the form:

ASSIGN *s* **TO** *i*

where *s* is the label of an executable statement in the current program unit and *i* is an integer variable name.

Execution of an ASSIGN statement causes the statement label (number) to be assigned to the integer variable (*i*). The variable is now defined for use only as a statement label reference; it is undefined as an integer variable. This statement label is required for referencing in an assigned GOTO statement or as a format identifier in an input/output statement.

For example:

```
ASSIGN 75 TO M
  ...
GOTO M
```

causes transfer of control to a statement with the label 75. Do not use the ASSIGN statement for arithmetic purpose, i.e., ASSIGN 75 to M is not equivalent to M = 75. If you define the integer variable with a statement label, you can redefine it with the same or a different statement label value or integer value. For example, using the statement:

```
M = 50
```

returns M to the status of an integer variable; it cannot be used in a GOTO statement.

Chapter 6
Control Statements

CONVEX FORTRAN usually executes statements in the order in which they are written. Control statements provide a means of altering the normal program execution sequence. The control statements include:

- GO TO—unconditional, computed, assigned
- IF—arithmetic, logical, block including ELSE IF, ELSE and END IF
- DO—indexed DO, *DO WHILE*
- *END DO*
- CONTINUE
- STOP
- PAUSE
- END

6.1 GOTO Statements

You can use the GOTO statements to change the flow of a program by transferring control to a specified statement. The three types of GOTO statements are unconditional GOTO, computed GOTO and assigned GOTO.

6.1.1 Unconditional GOTO Statement

The unconditional GOTO statement has the form:

GOTO *sl*

where *sl* is the label of an executable statement that appears within the current program unit.

During statement execution, control transfers to the statement identified by the statement label. The identified statement then executes. For example:

```
      GOTO 50
20    A = 5 * D
       ...
50    T = T + 1
```

transfers control to statement 50. In this example, to execute statement 20 and those statements immediately following it, control must transfer at some point to statement 20.

6.1.2 Computed GOTO Statement

The computed GOTO statement specifies the next executable statement from a list of several statements. The form of the computed GOTO statement is:

GOTO (*slist*) [,]*e*

where:

slist lists the labels of executable statements within the current program unit separated by commas (l_1,l_2 ...).

e is an arithmetic expression.

The computed GOTO statement evaluates the expression and transfers control to the labeled statement whose position in *slist* corresponds to *e*. If the value of *e* is less than 1 or greater than the number of statement labels in *slist*, control passes to the next statement in the program unit. For example, the statement:

```
GOTO (10,15,20,15,30)I
```

transfers control based upon the value of variable I. If I is 2, control passes to statement 15; if I is 5, control goes to statement 30. If the value of I is less than 1 or greater than 5, control passes to the first executable statement immediately following the computed GOTO.

As a CONVEX extension, e is converted to an integer data type when necessary. For example:

GOTO (10,15,20,15,30)X

6.1.3 Assigned GOTO Statement

The form of the assigned GOTO statement is:

GOTO *v* [[,] (*slist*)]

where:

v is an integer variable name defined by an ASSIGN statement with the value of an executable statement label.

slist is a list of one or more executable statements separated by commas (l_1,l_2,...) within the program unit.

When you execute an assigned GOTO, *v* must have as its value the value of a statement label attached to some executable statement in the same program unit. If several statement label values *slist* are present, the label assigned to *v* must be a member of that list. For example:

```
      ASSIGN 30 TO IFUN
10    GOTO IFUN (20,30,50)
        ...
30    FUN = 25.0 * 4.0
```

Statement 30 is executed immediately after statement 10. Control transfers to the statement label last assigned to *v* by the execution of a prior ASSIGN statement.

6.2 IF Statements

During program execution, IF statements permit transfer of control based on the value of an arithmetic or logical expression. The three types of IF statements are arithmetic, logical, and block.

6.2.1 Arithmetic IF Statement

The arithmetic IF statement evaluates the expression and transfers control based on the value of the expression. The arithmetic IF statement has the form:

IF (*ae*) l_1,l_2,l_3

where:

ae is an arithmetic expression.

l_1,l_2,l_3 are labels of executable statements contained within the current program unit. You must include all three labels for the statement to be valid.

During program execution the arithmetic expression is evaluated. If the value of the expression is less than zero, control transfers to the statement with the label l_1; to the statement with the label l_2 if the value is equal to zero; to the statement with the label l_3 if the value is greater than zero. For example, the statement:

```
IF (IA*IB) 40, 20, 50
```

transfers control to the statement with label 40 if the product of (IA*IB) is less than zero; to statement 20 if the product is zero; to statement 50 if the product is greater than zero. You can repeat the same statement label in the arithmetic IF statement.

For example, the following statement transfers control to statement 200 if MYEXAM is zero or greater than zero; if MYEXAM is less than zero, control transfers to statement 100.

```
IF (MYEXAM) 100, 200, 200
```

6.2.2 Logical IF Statement

The logical IF statement has the form:

IF (*le*) **es**

where:

le is a logical expression.

es is an executable statement except a DO, ***END DO***, END, logical IF, or a block IF statement. Do not use the statement to transfer control to any executable statement within a block IF statement or DO statement block.

The logical IF statement evaluates the value of the logical expression. If the value is .TRUE., the statement (*es*) is executed. After the statement is executed, control transfers to the next executable statement unless control is directed elsewhere by the statement (*es*). If the value evaluates to .FALSE., the next executable statement is executed.

Consider the following example:

```
IF (Y .AND. Q) Z = 7
IF (Y .LT. Q) GOTO 50
IF (Y .LE. Q) CALL SUB1
```

In the first statement, if both Y and Q are .TRUE., the value of Z is replaced by 7; otherwise, the value of Z remains unchanged. In the second statement, if the value of Y is less than Q, control transfers to the executable statement at 50; if Y is greater than Q, control transfers to the next executable statement. In the third statement, if the value of Y is less than or equal to Q, the subroutine SUB1 is called. If Y is greater than Q, control passes to the next executable statement; SUB1 is not called.

6.2.3 Block IF Statement

The block IF statement permits one or more statements (a block of statements) to be executed depending upon the value of the logical expression. The block begins with an IF THEN statement, followed by the statement block, and terminates with an END IF statement. The ELSE statement and the ELSE IF THEN statement can be included in the block IF statement.

There are several variations of the block IF statement. The basic form is:

IF (*le*) THEN
...
END IF

If *le* (logical expression) is true, all the lines (the block) between IF THEN and END IF are executed sequentially; otherwise, control transfers to the first executable statement following END IF. (You can put one or more statements in the block.)

Consider the following example:

```
IF (H .LE. 40) THEN
   P = H * PR
END IF
```

if H is equal to or less than 40, the block of statement(s), i.e., P = H * PR, is executed. After execution of the block, control transfers to the first executable statement following END IF.

Another variation of the block IF statement has the form:

IF (*le*) THEN
.
.
(*executable statements for true value*)
.
ELSE
.
.
(*executable statements for false value*)
.
END IF

If the expression (*le*) is true, the first block of statements is executed and the block following the ELSE statement is ignored. Control then passes to the END IF statement which transfers control to the next executable statement. However, if the expression (*le*) is false, the IF THEN block is skipped and control passes to the ELSE statement. Thus, the ELSE statement (block) is executed

only if the logical expression (*le*) of the block IF statement is false. For example, in the statements:

```
IF (H .LE. 40) THEN
    P = H * PR
  ELSE
    O = (H - 40) *PR *1.5
    P = H * PR + O
END IF
```

control transfers to the ELSE statement if H is more than 40. The ELSE block executes and control passes to the END IF which transfers control to the next executable statement unless the ELSE block has passed control elsewhere. However, if H is less than or equal to 40, the IF THEN block executes; the ELSE statement block is skipped.

A more complex IF block statement has the form:

IF (le_1) **THEN**
...
ELSE IF (le_2) **THEN**
...
ELSE IF (le_n) **THEN**
...
ELSE (optional depending upon program)
...
END IF

This IF block version allows you to use any number of additional logical expressions (*le*) to be evaluated if the value of the first logical expression (le_1) is false. The ELSE IF THEN statement executes if no preceding statement block in the IF block has executed and the value of the ELSE IF THEN is true. If both le_1 and le_2 are false, those block sequences are skipped and control transfers to the next ELSE IF THEN which is true. You can use any number of ELSE IF THEN statements. Use of the ELSE statement is optional, but you must include the ELSE statement if an action must be taken if all ELSE IF THEN statements evaluate to the value of false. You must conclude the block with the END IF statement. The next example contains an IF block with ELSE IF THEN and ELSE statements.

```
IF (N .LE. J) THEN
  K = M
    ELSE IF (N .GT. J/3) THEN
      K = I
    ELSE IF (N .EQ. J/3) THEN
      K =  -(MY)
    ELSE
      K = L
END IF
```

The IF block is evaluated sequentially; evaluation of each ELSE IF THEN statement continues until a true value is determined. Then that ELSE IF THEN statement executes and control transfers to the END IF statement which transfers control to the next executable statement. If all ELSE IF THEN statements evaluate as false, the ELSE statement executes.

6.2.3.1 Nested Block IF Statements

The initial block IF can contain nested block IF statements as long as the nested block IF is completely contained within a statement block. Each block begins with ane IF THEN statement and ends with an END IF statement. For example:

```
IF (T .GT. 40) THEN
    0 = X * 1.5
    IF (AT .GT. 60) THEN
      B = 25
    ELSE
      B = 0
    END IF
ELSE
    NP = H * P
END IF
```

If T is greater than 40, the block executes (0=X*1.5). Then the nested IF THEN is evaluated and executed according to the value of AT. If AT is greater than 60, the block executes. If AT is less than or equal to 60, control transfers to the ELSE statement. (The nested block IF must have an END IF.) If T is less than or equal to 40, the nested IF block does not execute. Control transfers to the outer ELSE statement.

6.3 DO Statements

A DO statement is used to specify a DO loop. The group of statements must follow the DO statement, be located within the program unit, and end with a terminal statement. You cannot transfer control into the range of a DO from elsewhere in the program unit, but you can terminate execution of a DO loop by transferring control outside the loop. The DO statement has the form:

DO [*sl* [,]] $v = ex_1, ex_2, ex_3$

where:

sl is the label of an executable statement (followed by an optional comma) in the current program unit. ***CONVEX FORTRAN does not require this label. If you do not specify a label, the DO loop must terminate with an END DO statement. That is, a statement of the form "DO I=1,100,2" must terminate with "END DO"***. Nested DO loops cannot share an unlabeled ***END DO*** statement, but they may share a labeled terminal statement.

The label identifies the last statement (terminal statement) of the DO loop and the label must textually follow the DO statement. You cannot use an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement as the terminal statement. You can use a logical IF statement as the terminal statement when the IF statement does not contain a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

v is the name of the integer, real or double-precision variable called the DO-variable.

ex_1 is an integer, real, or double-precision expression that specifies the beginning value of v upon the initial execution of the DO loop. (Each succeeding value is determined by ex_1+ex_3, ex_1+2*ex_3, ex_1+3*ex_3 ... until the value exceeds ex_2.)

ex_2 is an integer, real, or double-precision expression that specifies the ending value of v.

ex_3 is an integer, real, or double-precision expression that indicates the increment for v after the first DO loop is executed. If you omit the increment value and comma, the value defaults to 1.

The DO statement executes a loop that begins at the DO and ends with the terminal statement. During execution, CONVEX FORTRAN evaluates ex_1, ex_2, ex_3 to determine the beginning value of v, to determine the final value of v and to set the iteration count. Each time the DO loop executes, the value of the increment expression (ex_3) is added to the DO variable, and the iteration count is reduced by 1. The DO loop executes until the value of v exceeds ex_2 or executes a final time when v equals ex_2.

Transfer of control outside the loop terminates execution of the loop with the value of v remaining at the value currently defined. When a DO loop terminates normally, the value of v is always greater than ex_2. For example, the statement:

```
DO 20 MYEXAM = 2,6,2
```

indicates execute the DO loop for MYEXAM taking the values 2, 4, and 6. The value of the DO variable on termination of the DO loop becomes 8.

The iteration count is determined by $INT((ex_2-ex_1+ex_3)/ex_3))$. The loop executes until the iteration equals zero. Normally a negative or zero value indicates that the DO loop is not executed. ***If you specify the -F66 compiler option, the body of the loop executes once even when the iteration count is zero or negative.*** Since internal representation of real numbers is not exact, using a real number for the DO variable may produce a count that is unexpected. You cannot redefine the DO variable (v) within the range of the DO loop. You can, however, alter the initial, terminal, and increment parameters within the loop without affecting the iteration count.

After execution of the DO loop, if there is no nesting of loops, control transfers to the first executable statement after the terminal statement. If there is nesting of DO loops and DO loops share a terminal statement, control transfers outward to the next most enclosing DO loop.

6.3.0.2 Nested DO Loops

You can nest DO loops if each inner DO loop is entirely within the range of the outer DO loop. The DO loop can be entered only through the DO statement. During execution, control can be transferred out of the range before execution is completed and then control returned within the range of the DO loop. You cannot, however, transfer control from an outer loop to an inner loop. ***DO loops can share terminal statements but not unlabeled END DO statements.*** If DO loops share terminal statements, a transfer to that statement can be made only from within the range of the innermost DO. For example:

```
      DO 10 I = 1,100
       ...
      X = X + 1
      IF (X .GT. XMAX) GOTO 10
       ...
10    CONTINUE
```

6.3.0.3 Extended Range

To maintain compatibility with older FORTRAN implementations, CONVEX FORTRAN allows extended range DO loops. An extended range DO loop is a loop in which control transfers outside the body of the DO loop and then back into the loop. The statements in the extended range are logically in the body of the loop.

The extended-range DO statement control transfers work under the following rules (see Figure 6-1 for examples):

- *You may transfer into the range of a DO statement only if you make the transfer from the extended range of that same DO statement; the transfer is invalid otherwise.*
- *The extended range of a DO statement should not change its control variable.*

Figure 6–1: Valid and Invalid Control Transfers

Example of Valid Control Transfer Into Range of DO Statement From Its Extended Range:

```
  5    DO 10 I = 1,10
       IF (A(I).LE.0)GOTO 20        DO Loop
 30    A(I) = -1
 10    CONTINUE
       RETURN
 20    A(I) = F(X(I))               Extended Range
       GOTO 30
```

Example of Invalid Transfer Into Range of DO Statement:

```
       IF (FLAG)GOTO 5
       I = 3
       GOTO 30                      Invalid Control Transfer to DO Loop
  5    DO 10 I = 1,10
       IF (A(I).LE.0)GOTO 20
 30    A(I) = -1
 10    CONTINUE
 20    RETURN
```

6.3.1 DO WHILE Statement

The DO WHILE statement allows continual execution of a DO loop as long as a logical expression contained in the statement itself remains true. This statement has the form:

DO [s [,]] WHILE (e)

where:

s *is the label of an executable statement that must physically follow in the same program unit.*

e *is a logical expression.*

The DO WHILE statement checks the value of the logical expression at the beginning of each execution of the loop, starting with the first. When the value is true, execution of the statements in the loop follows; when the value is false, control transfers to the statement following the loop. If you do not include a label in the DO WHILE statement, you must terminate the DO WHILE loop with an END DO statement. An example of the use of the DO WHILE statement follows:

```
DO WHILE (ARRAY (I,J).GT.1.0)
   ARRAY (I,J) = ARRAY(I,J)/2.0
   I = I+1
   J = J-1
END DO
```

The condition is only tested at the top of the loop. If the condition occurs during the execution of the loop, the loop does not terminate until control passes back to the top of the loop.

6.4 END DO Statement

The END DO statement terminates the DO and DO WHILE statements. You must include the END DO statement at the end of a DO loop if the DO or DO WHILE statement defining the loop does not contain a terminal-statement label. You may also use the END DO statement as a labeled terminal statement if the DO or DO WHILE statement does contain a terminal-statement label. An example of how the END DO statement is used in context follows:

```
   REAL A(101)
   DO 10 I = 1,100
      A(I) = SIN(A(I))/COS(A(I+1))
10 END DO

   REAL A(101)
   DO I=1,100
      A(I)=SIN(A(I))/COS(A(I+1))
   END DO
```

6.5 CONTINUE Statement

The CONTINUE statement ends a DO loop and transfers control to the next executable statement. When used with a DO loop, a CONTINUE statement must be labeled; otherwise, no label is required. Because execution of a CONTINUE statement has no effect, you can place it anywhere in a program that an executable statement is allowed. This statement has the form:

```
CONTINUE
```

6.6 STOP Statement

The STOP statement terminates program execution and has the following form:

STOP [*s*]

where *s* is a string of five or fewer digits or a character constant to be displayed when the STOP statement executes.

Example:

```
STOP '-JOB FINISHED'
```

6.7 PAUSE Statement

Use the PAUSE statement to temporarily halt program execution. The program resumes action when the operator commands; the program itself does not control execution. The statement form is:

PAUSE [*s*]

where *s* is a string of five or fewer digits or a character constant.

For example, the statement:

PAUSE 'INSERT TAPE 1; PRESS RETURN TO CONTINUE'

temporarily halts program execution. After the tape is inserted and carriage return is pressed, program execution continues.

6.8 END Statement

The END statement terminates a main program without a message. In a function or subroutine subprogram, it returns control to the calling program performing the same function as a RETURN statement in a subprogram. It has the form:

END

The END statement must end every program unit and can appear only in columns 7 through 72 of an initial line.

Chapter 7
Input/Output Statements

Input/output statements provide a method for transferring data between internal storage and external media or between internal storage and internal files. CONVEX FORTRAN supports READ, *ACCEPT, and DECODE* statements for input, and WRITE, *TYPE*, PRINT, and *ENCODE* statements for output. Table 7-1 lists the supported I/O statements by category.

The auxiliary statements control the connection of files to external devices, position files, or retrieve information about a file or unit. These statements are: OPEN, CLOSE, REWIND, INQUIRE, BACKSPACE, ENDFILE, and *FIND*.

Table 7-1: Data Transfer I/O Statements

Statement Category	Statement Name						
	READ	WRITE	*ACCEPT*	*TYPE*	PRINT	*DECODE*	*ENCODE*
Sequential/ External:							
Formatted	Yes	Yes	*Yes*	*Yes*	Yes	*No*	*No*
Unformatted	Yes	Yes	*No*	*No*	No	*No*	*No*
List-Directed	Yes	Yes	*Yes*	*Yes*	Yes	*No*	*No*
Namelist-Directed	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*No*
Sequential/ Internal:							
Formatted	Yes	Yes	*No*	*No*	No	*Yes*	*Yes*
List-Directed	*Yes*	*Yes*	*No*	*No*	*No*	*No*	*No*
Direct/ External:							
Formatted	Yes	Yes	*No*	*No*	No	*No*	*No*
Unformatted	Yes	Yes	*No*	*No*	No	*No*	*No*
Direct/ Internal:							
Formatted	*Yes*	*Yes*	*No*	*No*	*No*	*No*	*No*

NOTE: All unformatted, internal I/O statements, all direct, list-directed, and all direct, namelist-directed I/O statements are not allowed. All other variations are allowed.

7.1 Records

A sequence of characters or values processed as a unit constitutes a record; I/O statements transfer all data as records. Formatted records contain characters; unformatted records (those written without format specification) consist of bytes that represent binary values. Each unformatted I/O statement transfers one record; formatted, list-directed, and namelist-directed I/O statements can transfer multiple records. Each list-directed and namelist-directed I/O statement transfers as many records as is required by the I/O data list. Each read or write begins with a new record.

7.1.1 Formatted Records

A formatted record consists of a sequence of characters (letters, numbers, and special symbols). You cannot use formatted I/O on files connected for unformatted access.

With formatted input, if the input statement requires more fields than are available, the fields are read as spaces. If the input statement does not require all the fields in the record, unneeded fields are ignored.

The processor reads or writes the current record and possibly additional records during data transfer. The length of the record is measured in characters and depends on the number of characters written to the record. The length can be zero. Any record values left unfilled during data transfer to fixed-length records are written as spaces. An error condition occurs if the size of the data is greater than the record length when an output statement writes to a fixed-length record.

7.1.2 Unformatted Records

An unformatted record consists of a sequence of zero or more bytes. You cannot use unformatted I/O on files connected for formatted I/O. For each unformatted I/O statement the processor reads or writes one record.

The number of bytes written determines the length of the unformatted record; the length can be zero. On input, if the data list requires more fields than are available, an error condition occurs. For fixed-length records, the data list in the output statement must not specify more values than the record can hold. Any record fields left unfilled during data transfer to fixed-length records become zeros.

7.1.3 ENDFILE Record

The ENDFILE statement writes the endfile record that ends the file. An endfile record is written when the file is closed, either through the CLOSE statement, a REWIND statment, or implicitly through program termination. The endfile record appears only as the last record of a file. You cannot use an ENDFILE statement when a file is connected for direct access.

7.2 Files

A sequence of records that are input to or output from a program constitutes a file. A file is either internal (array or variable) or external (located on a peripheral device). There are two methods of accessing files: sequential and direct.

7.2.1 Internal Files

An internal file consists of a character variable, array, array element, or substring into which records are read or written. If the file consists of a character variable, array element, or substring, it constitutes a single record. ***When the file consists of an array, each element constitutes a record.*** The internal file provides for transfer and conversion of data from internal storage to internal storage.

A record in the internal file can be read only if the record is defined. When the processor writes the record, the record of the internal file becomes defined. Also, you can use character assignment statements to define a record.

You can specify an internal file only in READ, WRITE, ***ENCODE, and DECODE*** statements.

7.3 Units

Before you can access an external file, you must associate (connect) it with a unit. Executing the OPEN statement accomplishes the connection by assigning a logical number to the external file. This number becomes the unit and file designator, which provides a means for referencing the file. Internal files are not connected or opened but are referenced by variable, array, or substring name. Connection also may be accomplished implicitly by the system. You cannot connect a file to more than one unit at a time. You can, however, connect a unit to a file that does not exist, that is, a new file that has not been written.

The following statements illustrate various ways to open a file. For instance, the statement:

```
OPEN (7)
```

opens the file *fort.7*; this is the file associated with unit 7 by default. Whereas, the following statement:

```
OPEN (8, FILE='TEST.DAT')
```

connects unit 8 to the file TEST.DAT.

The statement

```
OPEN (9, STATUS='SCRATCH')
```

opens a scratch (temporary) file associated with unit 9. When the file is closed or the program terminates, the file is deleted.

To reassign the unit, terminate the connection. A CLOSE statement (or an OPEN statement for another file) terminates the connection. The connection is terminated implicitly when the program ends.

7.4 Accessing Files

You can use either the sequential or the direct method for accessing records of a file. Connection of a file to a unit, typically accomplished with an OPEN statement, determines the method of access.

7.4.1 Sequential Access

To connect a file for sequential access use the OPEN statement. For example:

```
OPEN (10,FILE='MYEXAM',ACCESS='SEQUENTIAL')
OPEN (10,FILE='MYEXAM')
```

If you do not specify the ACCESS keyword, the access mode defaults to SEQUENTIAL. To change the access method, close the file and reopen it with the specification for the new access method.

A file connected for sequential access cannot be read/written with direct access I/O statements. A data-transfer statement causes the next record to be read or written when a file is connected for sequential access; the records are accessed in their order of placement in the file. The last record must be an endfile record.

7.4.2 Direct Access

If you connect a file for direct access, the records can be written or read in any order (randomly). The record number specified in the I/O transfer statement determines the order of processing. When you connect a file for direct access, you may not read/write the file with sequential-access I/O statements.

To establish a direct-access file, open a unit for direct access; for example:

```
OPEN (10,FILE='MYEXAM',ACCESS='DIRECT',RECL=1024)
```

All records of a direct-access file have the same length. The record size is specified in bytes when the file is opened.

7.5 I/O Statement Format

The general format of an I/O transfer statement is:

READ (*clist*) *iolist*
WRITE (*clist*) *iolist*

where:

clist is the control information list that controls the transfer of the data.

iolist is the I/O list that specifies the data to be transferred.

If invalid data is encountered in a READ statement, execution stops at that point and the remaining variables in the *iolist* are ignored.

7.5.1 Input/Output Lists

The I/O lists (*iolist*) identify the entities whose values are transferred by I/O data-transfer statements. An *iolist* entity can be:

- Character substring name (CHAR(6:10))
- Variable name (L)
- Array name (MYEXAM)
- Array element name (M(3))
- Implied-DO list (J,K,L,M, I=1,4)

- An expression (K + L or 'JKL'); used for output only. The expressions cannot contain function references with I/O statements in them.

When an array name without a subscript appears in an *iolist*, the elements are processed in the order in which they are stored, such as M(1,1), M(2,1) ....

7.5.2 Implied-DO Lists

An implied-DO list is used for specifying repetition of part of an I/O list, transferring part of any array, and transferring array elements in an order that is not the same as the order in which they are stored. The implied-DO loop has the form:

(*dlist*, *v*=*ae1*,*ae2* [,*ae3*])

where:

dlist	is an I/O list.
v	is an integer or real variable.
ae1,*ae2*,*ae3*	are arithmetic expressions.

The variable and arithmetic expressions have the same forms and functions as those in the standard DO statements. The loop begins with the value of *ae1* and increments by the value of *ae3* until it equals or exceeds the value of *ae2*, the loop then exits. Elements in *dlist* can reference *v*, but cannot change the value of *v*. The implied-DO loop can be nested.

The following statements illustrate uses of the implied-DO loop:

```
WRITE (7) (A,B,I=1,10)                 ! writes the pair A,B 10 times

READ (7) (A (I),I=5,10)                ! reads elements 5 through 10 of array A

WRITE (7) ( (A (I,J),J=1,N),I=1,N)     ! writes the array A by rows
```

7.6 Specifiers

You can use seven specifiers in the control information list to provide information on various aspects of data transfer. Each specifier consists of a keyword, an equal sign, and a parameter for the specifier. The specifiers are:

- Unit
- Format
- Record
- Status
- Error
- End of file
- *Namelist*

7.6.1 Unit Specifier

The unit specifier identifies the external or internal unit being accessed. It has the form:

[UNIT =] *u*

where *u* is an internal or external identifier. As an external file identifier, *u* is an integer of positive or zero value from 0 to 255 or ***, which defaults to a preassigned input or output unit.*** As an internal file identifier, *u* is the name of a character variable, array, array element, or substring.

The keyword UNIT= is optional if the unit specifier is the first item in a list of specifiers.

7.6.2 Format Specifier

You must include a format specifier in each data transfer statement to or from a formatted file. The format specifier is a label of a FORMAT statement, a character expression within the transfer statement, or an asterisk indicating list-directed formatting. A format specifier has the form:

[FMT =]*f* or **[FMT =]***

where:

f is a character expression (character constant or name of a character variable, array element, or substring) that contains a runtime format, a statement label of a FORMAT statement, or an integer variable with an assigned FORMAT statement label. The FORMAT statement must be in the current program unit.

* indicates list-directed formatting that uses default formatting based on the I/O list data types.

If the first item of the control information list is the unit specifier (without the keyword UNIT=) and the second item is the format specifier, you can omit FMT= from the format specifier. If no format specifier is included, the I/O statement is unformatted.

7.6.3 Record Specifier

The record specifier, when used in a data-transfer statement, indicates which record is to be read or written in a file connected for direct access. You may not use the record specifier for sequentially accessed files. A record specifier has the forms:

REC = *r* or *'r*

where *r* is a numeric expression with a positive value that specifies the position of the record to be accessed for I/O. ***If the second form is used, the unit specifier cannot use the UNIT keyword and the value for r must appear immediately after the unit specifier with no intervening comma, for example, WRITE (5'10). The second form is valid only if the -vfc compiler option is specified.***

7.6.4 Status Specifier

The status specifier provides a means for determining an error or end-of-file condition. A status specifier has the form:

IOSTAT = *ios*

where *ios* is an integer variable or array element.

After the I/O statement containing the status specifier executes, the status variable contains:

- Positive integer, which indicates an error condition exists; this integer is the error number.
- 0, which indicates normal execution; no error or end-of-file condition exists.
- -1, which indicates end-of-file condition.

If you indicate only the status specifier (no END or ERR specifier) in the statement and an error condition exists or an end-of-file condition occurs during program execution, program execution continues at the next executable statement.

7.6.5 Error Specifier

If an error occurs during program execution, an error specifier causes control to be transferred to the designated statement. An error specifier has the form:

ERR = *s*

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

When the processor detects an error during program execution, the I/O statement terminates immediately. The value of the status specifier (if included) becomes a positive integer and control transfers to the statement whose label appears in the error specifier.

7.6.6 End-of-File Specifier

You can use the end-of-file specifier in a statement when you want to transfer control to a specific statement on an end-of-file condition. An end-of-file specifier has the form:

END = *s*

where *s* is the statement label of an executable statement in the same program unit as the end-of-file specifier.

An end-of-file condition exists when the end-of-file record is read in an external file opened for sequential access, or when an attempt is made to read a record beyond the range of an internal file. When an end-of-file condition is detected during program execution, the READ statement terminates, the value of the status specifier (if included) becomes -1, and control transfers to the statement whose label appears in the end-of-file specifier.

7.6.7 Namelist Specifier

The namelist specifier allows you to specify that namelist-directed I/O is to be used and to specify the groupname for the entities that are modified during input or written on output. The namelist specifier has the form:

NML = nlgrpname

where nlgrpname is the symbolic name that has been defined for the entities in a NAMELIST statement.

If the first item of the control information list is the unit specifier without the keyword UNIT=, you can omit the keyword NML= from the namelist specifier, but you must place the namelist specifier (nlgrpname) as the second item in the control information list. Otherwise, you must always use the keyword NML=. You cannot use the namelist specifier in a statement containing a format specifier.

7.7 READ Statement

READ is an input statement that assigns values from a record to the *iolist* variables. Execution of the READ statement with an external file causes input data to be transferred from the external file into internal storage or memory. Execution of the READ statement with an internal file causes data to be transferred between internal storage locations.

The statement has the form:

READ (*clist*) [*iolist*]
READ *f* [,*iolist*]

where:

clist is a control information list (described in the "Specifiers" section). You must include a unit specifier in the READ statement *clist*, and if the record is formatted, a format specifier. The record specifier must be included for direct access of a file. You must include the namelist specifier for namelist-directed I/O. The status, error, and end-of-file specifiers are optional.

iolist is the I/O list that identifies the data to be transferred. The entities include variables, array elements, substrings, implied-DO lists, and/or array names.

f is the format specifier. The specifier is a character array name, a character expression, a character constant, statement label of a FORMAT statement, or an integer variable assigned the label of the FORMAT statement. Use an asterisk (*) to indicate list-directed formatting.

When the READ statement executes, at least one record consisting of values from the I/O entities is written. The file is then positioned at the beginning of the next record. CONVEX FORTRAN reads a sequential, formatted file unless an OPEN statement contains FORM='UNFORMATTED' and/or ACCESS='DIRECT'. READ statements that specify unformatted reads to internal files are not permitted. Direct-access, internal I/O is permitted. The logical record length is the length of the array element. Thus, a character variable array is similar to a fixed-length, direct-access file and obeys the same rules for formatted I/O.

The values transfer as one record unless you include a slash, which indicates a new record, in the format specification.

7.7.1 External Sequential READ Statements

There are four classes of external sequential READ statements: formatted, unformatted, list-directed and namelist-directed. The use of IOSTAT, ERR, and END status specifiers is optional in all four classes of statements.

For example, the formatted sequential READ statement, which requires a unit (*u*) and a format specifier (*f*), has the form:

READ (*u*,*f* [,IOSTAT,ERR,END]) [*iolist*]

For example:

```
READ (UNIT=50, FMT=10, IOSTAT=IOERR, ERR=120) D, E, F
READ (50, 10) D, E, F
```

Both statements sequentially read values into D, E, and F according to the format specified by statement 10. The first READ statement returns any error codes in the variable IOERR and transfers control to statement 120 on an error condition.

An unformatted sequential READ statement, which requires a unit specifier (*u*), has the form:

READ (*u* [,IOSTAT,ERR,END]) [*iolist*]

For example:

```
READ (UNIT=*, END=260) D, E
READ (50)
```

The first statement sequentially reads from the implicit input unit values into D and E without any conversion. The second statement skips the next record in the file connected to unit 50.

A list-directed sequential-access statement, which must contain * to indicate list-directed formatting, has the form:

READ (*u*,* [,IOSTAT,ERR,END]) [*iolist*]
READ *[,*iolist*]

For example:

```
READ (UNIT=50, FMT=*) D, E, F
READ (50, *, IOSTAT=IOERR)
READ *, D, E, F
```

The first statement assigns values to D, E, and F from the current record of the file connected to unit 50. Conversion from ASCII to internal format is done according to the rules for list-directed formatting. The second statement skips the current record of the file connected to unit 50. The last statement reads from the implicit input unit into the variables D, E, and F under list-directed formatting.

The namelist-directed sequential READ, with a unit specifier (u) and a namelist specifier (nl) in the control information list, has the form:

READ (u,nl [,IOSTAT,ERR,END])

When the namelist-directed READ is used with a nonkeyword (no control information list), it has the form:

READ nlgrpname

where nlgrpname represents the name associated with a list of entities.

When you use the namelist-directed READ statement, you must have a NAMELIST statement in the program segment.

For example:

NAMELIST /SAM/ NAME, EXAM1, EXAM2, EXAM3
*CHARACTER*5 NAME*
READ (UNIT=50, NML=SAM) or READ SAM

*The first statement associates the name (SAM) with the four entities. The second statement defines NAME to be a CHARACTER*5 variable; EXAM1, EXAM2, and EXAM3 are implicitly typed. The third statement reads input data and assigns values to the namelist entities—NAME,EXAM1, EXAM2, EXAM3. The READ statement reads data until it finds the specified name (SAM). Then it translates the data from external to internal form, using the data type of the entities and the form of the input. Then the translated data is assigned to the specified entities (NAME, EXAM1, EXAM2, EXAM3) in the order they appear in the input records. (See Chapter 8 for detailed information on inputting values.)*

7.7.2 External Direct READ Statements

There are two classes of external direct-access READ statements: formatted and unformatted.

A formatted direct-access statement must contain a unit specifier (*u*), record number (*rn*), and format specifier (*f*). This statement has the form:

READ (*u,rn,f* [**,IOSTAT,ERR,END**]) [*iolist*]

For example:

```
READ (119, 100, REC=25) D, E, F
```

This statement reads record number 25 of the file connected to unit 119 and assigns values to D, E, and F from this record.

An unformatted direct-access statement, which must contain a unit (*u*) and record (*rn*) specifier, has the form:

READ (*u,rn* [**,IOSTAT,ERR,END**]) [*iolist*]

For example:

```
READ (50, REC=1) D, E, F
```

In this case, the statement reads the first record of the file connected to unit 50 and assigns values from it without translation to the variables D, E, and F.

7.7.3 Internal READ Statements

The internal READ statement transfers and converts information from internal storage. In the internal READ statement, the name of the character variable, array, array element, or substring (*iu*) is used in place of the external identifier. The use of IOSTAT, ERR, and END status specifiers is optional.

The internal sequential-access READ statement is always formatted and has the form:

READ (*iu,f* [,IOSTAT,ERR,END]) [*iolist*]

The following statement transfers values from MYEXAM to A and B, converting them from ASCII to internal form according to the format at line 25.

```
READ (MYEXAM,25) A, B
```

The following statement uses list-directed formatting.

READ (MYEXAM,) A, B*

The internal direct-access READ statement has the form:

READ (u,f,rn, [IOSTAT,ERR,END]) [iolist]

For example:

READ (ARR, 10, REC=2) A

This statement converts the second element of the array ARR from ASCII to internal form and stores the result in A. The logical record length is the length of the array element. Thus, a character variable array is similar to a fixed-length, direct-access file, and follows the same rules.

7.8 ACCEPT Statement

The ACCEPT statement sequentially reads data from the implicit input unit and has the following formats:

ACCEPT f [,iolist]
*ACCEPT * [,iolist]*
ACCEPT nlgrpname

where:

f	*is the nonkeyword form of a format specifier.*
*	*specifies list-directed formatting.*
iolist	*is an I/O list.*
nlgrpname	*is the nonkeyword form of the namelist specifier.*

The ACCEPT statement is like the READ formatted or list-directed, sequential, external statement except that reading is always done from the implicit input unit.

Example:

```
     ACCEPT 100, I, J
100  FORMAT (2I)
```

As shown, the ACCEPT statement reads integer data from the implicit input unit and assigns values to the integer variables I and J.

7.9 WRITE Statement

The WRITE statement transfers data from internal storage to external devices or from internal storage to internal files. The WRITE statement has the form:

WRITE (*clist*) (*iolist*)

where:

clist is a control information list (described in the "Specifiers" section). that must include a unit specifier. A formatted record must include a format specifier. A record specifier must be included for direct access of a file. The status and error specifiers are optional. (End-of-file is not allowed in WRITE statements.)

iolist is the I/O list that identifies the data to be transferred. The entities may include variables, array elements, substrings, implied DO lists and/or array names, and expressions.

f is the format specifier and is a character array name, a character expression, a character constant, statement label of a FORMAT statement, or an integer variable assigned the label of the FORMAT statement. An asterisk (*) indicates list-directed formatting.

The WRITE statement writes at least one record consisting of values from the I/O entities. The file is then positioned at the beginning of the next record. CONVEX FORTRAN writes a sequential, formatted file unless an OPEN statement contains FORM='UNFORMATTED' and/or specifies ACCESS='DIRECT'. Unformatted writes to internal files are not permitted. Direct-access, internal I/O is permitted. The logical record length is the length of the array element. A character variable array is similar to a fixed-length, direct-access file, and obeys the same rules for formatted I/O.

The values transfer as one record unless you include a slash, which indicates a new record, in the format specification. If the WRITE statement specifies I/O to a nonexistent file, the file is created unless an error condition occurs.

7.9.1 Sequential-Access WRITE Statements

There are four classes of sequential WRITE statements—formatted, unformatted, list-directed, and namelist-directed. The use of IOSTAT and ERR specifiers is allowed in all four classes of statements.

For example, the formatted sequential WRITE statement, which requires a unit (*u*) and a format (*f*) specifier, has the form:

WRITE (*u*,*f* [,**IOSTAT**,**ERR**]) [*iolist*]

For example:

```
WRITE (UNIT=50, FMT=10, IOSTAT=IOERR, ERR=120) D, E, F
WRITE (50, 10) D, E, F
```

Both statements transfer formatted values from the file connected to unit 50 to the variables D, E, and F. The first statement, however, allows for transfer of control if an error condition exists. In this example, if an error condition exists, the error number is assigned to IOERR and control transfers to statement 120.

An unformatted sequential WRITE statement, which requires a unit (*u*) specifier, has the form:

WRITE (*u* **[,IOSTAT,ERR]**) [*iolist*]

For example:

```
WRITE (UNIT=50) D, E
WRITE (50)
```

The first statement writes two unformatted values to unit 50. The second writes an empty record to unit 50.

A list-directed sequential-access WRITE statement, which must contain * to indicate list-directed formatting and a unit (*u*) specifier, has the form:

WRITE (*u*,* **[,IOSTAT,ERR]**) [*iolist*]

For example:

```
WRITE (UNIT=50, FMT=*) D, E, F
```

which writes D, E, and F according to the default format used for list-directed I/O.

The namelist-directed WRITE statement, which requires a unit (u) and a namelist (nl) specifier, has the form:

WRITE (u,nl [,IOSTAT,ERR])

In the following example, the statement transfers data from the variables specified by the namelist specifier SAMPLE to the file connected to unit 50.

WRITE (UNIT=50, NML=SAMPLE)

7.9.2 Direct-Access WRITE Statements

There are two classes of external direct-access WRITE statements—formatted and unformatted. The use of IOSTAT and ERR status specifiers is optional in both statement classes.

For example, a formatted direct-access statement, which must contain a unit (*u*) specifier, record number (*rn*), and format (*f*) specifier has the form:

WRITE (*u,rn,f* **[,IOSTAT,ERR]**) [*iolist*]

For example:

```
WRITE (50, 100, REC=25) D, E, F
WRITE (50, 100, REC=25, ERR=100) D, E, F
```

Both statements write D, E, and F to record number 25 of unit 50 according to the format specified in statement 100. The second statement also transfers control to statement 100 if an error condition exists. An unformatted direct-access statement, which must contain a unit (*u*) and record number (*rn*) specifier, has the form:

WRITE (*u,rn* [,**IOSTAT,ERR**]) [*iolist*]

For example:

```
WRITE (50, REC=25) D, E, F
WRITE (50, REC=25, ERR=250) D, E, F
```

Both statements write D, E, and F to record number 25; no data formatting occurs. The second statement transfers control to statement number 250 if an error condition exists.

7.9.3 Internal Direct-Access WRITE Statements

The internal WRITE statment converts data from one location in memory to another. In the internal WRITE statement, the name of the character variable, array, array element, or substring (*iu*) is used in place of the external unit identifier. The use of IOSTAT and ERR status specifiers is optional. The internal sequential-access WRITE statement is always formatted and has the form:

WRITE (*iu,f* [,**IOSTAT,ERR**]) [*iolist*]

Some examples of valid internal direct-access write statements are:

```
WRITE (MYEXAM,25) A, B
WRITE (MYEXAM,*) A, B
```

which transfer values from A and B to MYEXAM, converting them from internal form to ASCII. The first example uses a format at statement 25, while ***the second statement uses list-directed formatting.***

The internal direct-access WRITE statement has the form:

WRITE (iu,f,rn, [IOSTAT,ERR,END]) [iolist]

For example:

```
WRITE (ARR, 10, REC=2) A
```

This statement transfers the values from A to the second element of ARR, converting them from internal form to ASCII according to the format specified at statement 10.

7.10 PRINT and TYPE Statements

You can use either the PRINT statement or the *TYPE* statement to transfer formatted records to the implicit output device. These statements use the sequential mode of access. The forms are:

PRINT *f* [,*iolist*] or *TYPE f [,iolist]*
PRINT * [,*iolist*] or *TYPE * [,iolist]*
PRINT *nlgrpname* or *TYPE nlgrpname*

where:

f is the format specifier.

* specifies list-directed formatting.

iolist is an I/O list.

nlgrpname *is the nonkeyword form of the namelist specifier.*

Example:

```
     CHARACTER*16 CLASS, RANK
     TYPE 400, CLASS, RANK
400  FORMAT ('CLASS=',A,'RANK=',A)
```

As shown, the TYPE statement writes one record to the implicit output device; the record consists of four fields of character data.

7.11 Additional Statements

The *ENCODE, DECODE, and FIND* statements are extensions to the ANSI standard and have been included to allow for compatibility with other FORTRAN versions and for ease in transporting older FORTRAN programs to CONVEX machines.

7.11.1 ENCODE Statement

The ENCODE statement is equivalent to the WRITE formatted, sequential, internal statement. ENCODE transfers data between arrays or variables in internal storage and translates the data from internal to character form. An example of the ENCODE statement follows:

ENCODE (c,f,b [,IOSTAT=ios] [,ERR=s]) [iolist]

where:

c *is an integer expression (the number of characters (bytes) to be translated to character form).*

f *identifies the format (an error results if you specify more than one record).*

b *is an array, array element, variable, or character substring reference, any of which receives the characters after translation to external form.*

ios *is either an integer array element or an integer variable that is defined as a positive integer if an error occurs and as a zero if no error occurs.*

s *is the label of an executable statement to which control transfers if an error occurs during I/O transfer.*

iolist *is an I/O list that contains the data to be translated to character form.*

The ENCODE statement translates the elements in the I/O list to character form, as specified by the format identifier and stores the characters in b. If the number of characters transferred is less than c, the remaining positions fill with blanks. If b is an array, its elements are processed in the order of subscript progression.

*The data type of b in any given statement determines the number of characters that the ENCODE statement processes. An array of LOGICAL*2, for example, can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array; a character array can contain characters equal in number to the length of each element multiplied by the number of elements; a character variable or character array element can contain characters equal in number to its length.*

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Example:

```
      CHARACTER*8 A
      I=1000
      J=9
      ENCODE (8,100,A) I,J
  100 FORMAT(2I4)
```

Result:

```
  A='1000^^^9'
```

7.11.2 DECODE Statement

The DECODE statement is equivalent to the READ formatted, sequential, internal statement. DECODE transfers data between arrays or variables in internal storage and translates the data from character to internal form. The DECODE statement is represented as follows:

```
  DECODE (c,f,b [,IOSTAT=ios] [,ERR=s] ) [iolist]
```

where:

- *c* — *is an integer expression (the number of characters (bytes) to be translated to internal form).*
- *f* — *identifies the format (an error results if more than one record is specified).*
- *b* — *is an array, array element, variable, or character substring reference, any of which contains the characters to be translated to internal form.*
- *ios* — *is either an integer array element or an integer variable that is defined as a positive integer if an error occurs and as a zero if no error occurs.*
- *s* — *is the label of an executable statement.*
- *iolist* — *is an I/O list that receives the data after translation to internal form.*

The DECODE statement translates the character data in b to internal (binary) form according to the format specifier and stores the elements in the list. If b is an array, its elements are processed in the order of subscript progression.

*The data type of b in any given statement determines the number of characters that the DECODE statement processes. A LOGICAL*2, for example, can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array; a character array can contain characters equal in number to the length of each element multiplied by the number of elements; a character variable or character array element can contain characters equal in number to its length.*

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Example:

```
        CHARACTER*8 A
        DATA A/'1000^^^9'/
        DECODE(8,100,A) I,J
    100 FORMAT (2I4)
```

Result:

```
    I=1000
    J=9
```

7.11.3 FIND Statement

The FIND statement positions a direct-access file to a particular record. It also sets the associated variable of the file to that record number. No transfer of data occurs. The FIND statement is represented as:

```
    FIND ( [UNIT=] u,REC=r [,ERR=s] [,IOSTAT=ios] )
```

where:

u	*is a logical unit number; it must refer to a direct-access file.*
r	*is the direct-access record number; it cannot be less than 1 or greater than the number of records defined for the file.*
s	*is the label of the executable statement to which control transfers if an error occurs.*
ios	*is an integer variable or integer array element that is defined as a positive integer if an error occurs and as a zero if no error occurs.*

Example 1:

```
    FIND (2,REC=1)
```

This statement positions unit 2 to the first record of the file; the associated file variable is set to 1.

Example 2:

```
    FIND (4,REC=INDX)
```

This statement positions the file to the record identified by the content of INDX; the associated file variable is set to the value of INDX.

7.12 Auxiliary Input/Output Statements

Auxiliary statements control the connection of files to external devices, position files, or retrieve information about files or units. The auxiliary statements include:

- OPEN
- CLOSE
- INQUIRE
- REWIND
- BACKSPACE
- ENDFILE

7.12.1 OPEN Statement

The OPEN statement connects an existing external file to the specified unit, changes the attributes of a connected file, or creates a new file and connects it to the specified unit. The statement has the form:

OPEN (*specifier* [,*specifier*]...)

Each specifier normally consists of a keyword and its value. You must include a unit number in the OPEN statement; all other specifiers are optional. The specifiers may be listed in any order except that the unit number must be first when it is given without the UNIT= keyword. For example:

```
OPEN (7, FILE='TEST.DAT', RECORDTYPE='FIXED', RECL=80)
```

connects the file TEST.DAT to unit 7 and defines the file to be a sequentially accessed formatted file with fixed-length records of 80 characters.

The following sections describe the OPEN statement keywords, which are summarized in Table 7-2. In the descriptions, the term "numeric expression" can be any integer or real expression. The value of the expression is converted to INTEGER data type before it is used in the OPEN statement.

NOTE

The OPEN statement operates somewhat differently under COVUEshell. Please refer to the *CONVEX COVUEshell Reference Manual* for more details.

7.12.1.1 ACCESS Keyword

The ACCESS keyword indicates the method of file access—direct or sequential. ***'APPEND' implies sequential access with positioning after the last record in the file.*** You must include the record length, RECL, in the list when ACCESS='DIRECT'. The keyword has the form:

ACCESS = *cex*

where *cex* is a character expression 'DIRECT', 'SEQUENTIAL', *or 'APPEND'*. The default is 'SEQUENTIAL'.

The following statement opens the file *tst* for sequential access with positioning after the last record in the file.

```
OPEN (UNIT=10, FILE='tst', ACCESS='APPEND')
```

Table 7-2: OPEN Statement Keywords

Keyword	Values	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' *'APPEND'*	Access mode	'SEQUENTIAL'
ASSOCIATEVARIABLE	*V*	*Next direct-access record*	*None*
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	*E*	*Physical block size*	*System default*
CARRIAGECONTROL	*'FORTRAN'* *'LIST'* *'NONE'*	*Print control*	*'LIST' (Formatted)* *'NONE' (Unformatted)*
DEFAULTFILE	*C*	*Default file specification*	
DISPOSE *DISP*	*'KEEP' or* *'SAVE' or* *'DELETE'*	*File disposition at close*	*Depends on STATUS keyword*
ERR	S	Error transfer label	
FILE *NAME*	C	File-name specification	fort.n, where n is the unit number
FORM	'FORMATTED' *'PRINT'* 'UNFORMATTED'	Format type	'FORMATTED' for sequential access; 'UNFORMATTED' for direct access
IOSTAT	V	I/O status	
MAXREC	*E*	*Direct-access record limit*	*Unlimited*
NOSPANBLOCKS	*None allowed*	*Ignored--for VAX compatibility only*	
READONLY	-	*Write protection*	*Depends on access rights to file*
RECL *RECORDSIZE*	E	Record Length	As specified at file creation
RECORDTYPE	*'FIXED'* *'VARIABLE'*	*Record structure*	*'VARIABLE' for sequential access; 'FIXED' for direct access*
STATUS *TYPE*	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
UNIT	E	Logical unit number	

Key: E is a numeric expression.
C is a character expression, *numeric array name, numeric variable name, or numeric array element name.*
V is an integer variable name.
S is a statement label.

7.12.1.2 ASSOCIATEVARIABLE Keyword

The ASSOCIATEVARIABLE keyword specifies an integer variable to be updated after each direct access I/O operation. The keyword has the form:

ASSOCIATEVARIABLE = asv

where asv is an integer variable.

After each direct-access I/O operation, asv is set to the number of the next sequential record in the file. This identifier is valid for direct-access mode only; it is ignored for other access modes.

7.12.1.3 BLANK Keyword

The BLANK keyword determines the interpretation of blank characters in numeric formatted input fields. The keyword has the form:

BLANK = *blnk*

where *blnk* specifies the character expression 'NULL' or 'ZERO'. The default is 'NULL' *unless the -F66 compiler option is specified, in which case the default is 'ZERO'.*

If you specify BLANK='NULL', all blanks are ignored. When BLANK='ZERO', all blanks except leading blanks are read as zeros.

7.12.1.4 BLOCKSIZE Keyword

The BLOCKSIZE keyword specifies the physical transfer size (in bytes) for the file. The keyword has the form:

BLOCKSIZE = bls

where bls is a numeric expression.

The default is the system default for the device. If you specify BLOCKSIZE, the physical record for block devices is set to the value of bls, with a maximum of 64k bytes. For other devices, the BLOCKSIZE value is rounded up to a multiple of the file system block size.

The following statements write one physical record of 200 bytes to the block-mode tape device /dev/mt12. The physical record consists of two logical records, each 100 bytes long.

```
CHARACTER*1 A(100), B(100)
...
OPEN (7, FILE='/dev/mt12', BLOCKSIZE=200, RECORDTYPE='FIXED', RECL=100)
...
WRITE (7, '(100A1)') (A(I), I=1,100)
WRITE (7, '(100A1)') (B(I), I=1,100)
...
```

7.12.1.5 CARRIAGECONTROL Keyword

The CARRIAGECONTROL keyword determines the type of carriage control processing to be used when printing a file. The keyword has the form:

CARRIAGECONTROL = cc

where cc is a character expression having a value equal to 'FORTRAN', 'LIST', or 'NONE'.

The default is 'LIST' for formatted files and 'NONE' for unformatted files. the control characters required to interpret it on an ASCII output device. These control characters are Control-L for start of page, New Line for double spacing, and null (no character) for single spacing.

Files created with CARRIAGECONTROL='LIST' may be printed with the fpr utility.

7.12.1.6 DEFAULTFILE Keyword

The DEFAULTFILE keyword contains a default file specification. The keyword has the form:

DEFAULTFILE = C

where c is a character expression that specifies a default file specification.

7.12.1.7 DISPOSE Keyword

The DISPOSE keyword allows you to keep, save, or delete files connected to the unit when the unit is closed. The keyword has the form:

DISPOSE = dis
DISP = dis

where dis is a character expression having a value equal to 'KEEP', 'SAVE', or 'DELETE'.

Specifying 'KEEP' or 'SAVE' retains the file after the unit is closed; 'DELETE' causes the file to be deleted. For scratch files, the default is 'DELETE'. For all other files, the default is 'KEEP'. The preferred method of deleting a file, according to the ANSI standard, is to use the STATUS='DELETE' keyword in the CLOSE statement.

The following example causes the file associated with unit 10 to be deleted when closed.

OPEN (UNIT=10, DISP='DELETE')

7.12.1.8 ERR Keyword

The ERR keyword specifies a statement number to which control is passed if an error occurs during execution of the OPEN statement. The keyword has the form:

ERR = *sl*

where *sl* specifies the statement label of an executable statement that appears in the same program unit as the error specifier.

7.12.1.9 FILE Keyword

The FILE *(or NAME)* keyword specifies the name of the file being connected to the unit. The keyword has the form:

FILE = *fln or NAME = fln*

where *fln* represents a character expression, ***a numeric array name (if it is a packed array), a numeric variable name, or a numeric array element name.*** If the FILE specification does not appear in an OPEN statement, the unit is connected to a predefined file.

When the filename is stored in a numeric variable, numeric array, or numeric array element, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). If the filename is stored in a character variable, array, or array element, it must not contain a zero byte.

The following statement opens the file *tst.in* for sequential access and connects it to unit 1.

```
OPEN (UNIT=1, FILE='tst.in')
```

7.12.1.10 FORM Keyword

The FORM keyword indicates either formatted or unformatted I/O. The keyword has the form:

FORM = *f*

where *f* represents the character expression with the value 'FORMATTED', 'UNFORMATTED', *or 'PRINT'*. If you do not specify a format specifier in the OPEN statement, formatted I/O is assumed for sequentially accessed files. Unformatted I/O is assumed for direct-access files.

Specifying FORM='PRINT' implies "formatted" and carriage control = 'FORTRAN' and enables vertical format control for that unit. Vertical format control is interpreted at runtime only on sequential formatted writes to a PRINT file.

As an alternative to specifying FORM='PRINT', use the UNIX utility fpr before printing the file to interpret vertical format control.

7.12.1.11 IOSTAT Keyword

The IOSTAT keyword provides a variable that is set to indicate the status of OPEN operation. The keyword has the form:

IOSTAT = *ios*

where *ios* is an integer variable or integer array element. A nonzero value returned in *ios* indicates an error condition.

7.12.1.12 MAXREC Keyword

The MAXREC keyword determines the total number of records allowed in a direct-access file. The keyword has the form:

MAXREC = mr

where mr is a numeric expression.

The MAXREC keyword applies only to direct-access files. The default is an unlimited number of records.

The following statement opens the file associated with unit 1 for direct access. Records past record number 100 cannot be accessed.

OPEN (1, ACCESS='DIRECT', MAXREC=100)

7.12.1.13 NOSPANBLOCKS Keyword

The NOSPANBLOCKS keyword specifies that records are not to cross disk block boundaries. This keyword is provided for VAX compatibility only and is ignored by CONVEX FORTRAN. The keyword has the form:

NOSPANBLOCKS

7.12.1.14 READONLY Keyword

The READONLY keyword specifies that an existing file can be read but not written. The keyword has the form:

READONLY

Using READONLY in an OPEN statement does not prevent the file from being removed when it is closed.

7.12.1.15 RECL Keyword

The RECL *(or RECORDSIZE)* keyword specifies the record size for fixed-length records and the maximum record size for variable-length files. The keyword has the form:

RECL = *ie* *or RECORDSIZE = ie*

where *ie* is an integer expression with a positive value. You must specify RECL when the file is opened with *RECORDTYPE='FIXED'*. The record size is measured in bytes; the default is 80 bytes. For fixed-length records, RECL is the size of the logical record buffer. For variable-length records, RECL is an initial approximation of the logical record size and the buffer is incremented in multiples of RECL bytes.

The following statement opens the direct-access unformatted file connected to unit 10. Each record in the file is 20 bytes long.

```
OPEN (UNIT=10, RECL=20, ACCESS='DIRECT', FORM='UNFORMATTED')
```

7.12.1.16 RECORDTYPE Keyword

The RECORDTYPE keyword specifies fixed- or variable-length records for a file. The keyword has the form:

RECORDTYPE = typ

where typ is a character expression whose value is equal to 'FIXED' or 'VARIABLE'. The defaults are:

File Access	*Default Recordtype*
Direct	*Fixed*
Sequential	*Variable*
List-directed	*Variable*

If you specify RECORDTYPE='FIXED', you must also specify RECL. RECORDTYPE='VARIABLE' is not allowed for direct-access files.

The following statement specifies sequential-access, fixed-length records, each of which is 10 bytes long. The file is connected to logical unit 10.

OPEN (10, RECORDTYPE='FIXED', RECL=10)

7.12.1.17 STATUS Keyword

The STATUS *(or TYPE)* keyword determines the status of the file to be opened; the default is 'UNKNOWN'. The keyword has the form:

STATUS = *sta or TYPE = sta*

where *sta* is a character expression with the value of 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'.

If you specify STATUS='OLD', the file must exist. To create a new file, specify STATUS='NEW' in the OPEN statement. When 'SCRATCH' is designated as the status, the unit is connected to a predefined file and must not be named. When the CLOSE statement is executed, the 'SCRATCH' file is deleted. When STATUS='UNKNOWN', CONVEX FORTRAN searches the directories to see if the file exists. If it does, status becomes 'OLD'; if it does not exist, status becomes 'NEW'.

If the STATUS *(or TYPE)* keyword is not specified, by default, scratch files are deleted and all other files are retained.

7.12.1.18 UNIT Keyword

The UNIT keyword specifies the logical unit to which a file is to be connected. The keyword has the form:

[**UNIT=**] *u*

where *u* is a numeric expression. The valid logical unit numbers are 0-255.

If the unit number appears as the first parameter of the OPEN statement, the UNIT keyword may be omitted; if the unit number appears elsewhere in the OPEN statement, the UNIT keyword is required.

If a unit is connected to a file but the FILE= specifier does not appear in the OPEN statement, the file to be opened is the same as the file to which the unit is currently connected. In this case, the BLANK= and FORM= specifiers are the only specifiers that can have a value different from the one currently in effect. When the OPEN statement executes, the new value of the BLANK= specifier becomes effective. The position of the file is unaffected.

If the file to be opened is not the same as the file to which the unit is connected, the effect is the same as executing a CLOSE statement immediately prior to the execution of the OPEN statement.

If a file is connected to a unit, you cannot reopen the file with a different unit number.

7.12.2 CLOSE Statement

Use the CLOSE statement to disconnect a file from a unit. The CLOSE statement is represented as:

CLOSE ([UNIT=]*u* [,STATUS=*p*] [,ERR=*s*] [,IOSTAT= *ios*])
CLOSE ([UNIT=]*u* *[,DISPOSE=p]* [,ERR=*s*] [,IOSTAT=*ios*])
CLOSE ([UNIT=]*u* *[,DISP=p]* [,ERR=*s*] [,IOSTAT=*ios*])

where:

u is a logical unit number that must be an integer expression.

p is a character expression that determines the disposition of the file. Its values are 'KEEP', *'SAVE'*, or 'DELETE'.

s is the label of an executable statement.

ios is an integer variable or integer array element.

The statement:

```
CLOSE (7, STATUS='DELETE')
```

disconnects the file opened to unit 7 and deletes it. Specifying either *'SAVE'* or 'KEEP' retains the file after you close the unit. If the unit is not connected to a file, the CLOSE statement has no effect.

The status specification supercedes the disposition specified in the OPEN statement. For scratch files, the default is 'DELETE'. For all other files, it is 'KEEP'. If you disconnect a unit or file by the CLOSE statement, either can be connected again within the same executable program to the same file or unit.

7.12.3 INQUIRE Statement

The INQUIRE statement determines specific information about a file or unit, such as the access mode or blocksize. This statement has the forms:

INQUIRE (FILE=*fi*, *list*)
INQUIRE ([UNIT=]*u*, *list*)

where:

fi is a character expression, ***numeric array name, numeric variable name, or numeric array element name*** whose value is the name of the file being queried.

list is a list of specifiers that indicate the information to be determined for the file or unit. Each specifier appears in the list only once. Table 7-3 describes the valid specifiers.

u is the external unit specifier (number) that identifies the unit to be queried. The unit need not exist nor need it be connected to a file. If the unit is connected to a file, the inquiry includes both the connection and the file.

Although you can position FILE=*fi* and UNIT=*u* any place in the list that specifies properties, if you omit the UNIT keyword, *u* must be the first item in the list.

The following statement returns the access mode of the file connected to unit 99 in the character variable ACC.

```
INQUIRE (99, ACCESS=ACC)
```

The following statement returns the form of the file, 'FORMATTED' or 'UNFORMATTED', in the character variable FM.

```
INQUIRE (FILE='TEST.IN', FORM=FM)
```

Table 7-3: INQUIRE Specifiers

Specifier/Form	Specifier Variable Values
ACCESS = character*	'DIRECT' or 'SEQUENTIAL' if connected *'UNKNOWN' if no connection*
BLANK = character*	'NULL' or 'ZERO' if connected and formatted I/O 'UNKNOWN' if no connection and/or unformatted I/O
*BLOCKSIZE = integer**	*0 if not connected. Block size set on OPEN; system default if not set on OPEN*
*CARRIAGECONTROL = character**	*'FORTRAN' if FORTRAN specified on OPEN; 'LIST' if specified on OPEN; 'NONE' if specified on OPEN; 'UNKNOWN' if not connected*
DIRECT = character*	'YES' if direct access permitted; 'NO' if direct access not permitted; 'UNKNOWN' if not connected
ERR = statement label	Control transfers to statement if error condition
EXIST = logical*	*true* if by file and exists; *true* if by unit and unit is in allowed set of unit numbers; *false* otherwise
FORM = character*	'FORMATTED' if connected for formatted 'UNFORMATTED' if connected for unformatted *'UNKNOWN' if not connected*
FORMATTED = character*	'YES' if formatted I/O permitted 'NO' if formatted I/O not permitted 'UNKNOWN' if file not connected
IOSTAT = integer*	0 if no error condition; positive integer if error condition
NAME = character*	*filename* if file has name; blank if no filename
NAMED = logical*	*true* if file has a name; *false* if no name
NEXTREC = integer*	*next record number* if record length specified on OPEN *0 otherwise*
NUMBER = integer*	unit number of file connected; -1 if no unit connected
OPENED = logical*	*true* if file/unit connected *false* if file/unit not connected
RECL = integer*	*record length* set on OPEN if connected for direct access; 0 otherwise
*RECORDTYPE = character**	*'FIXED' if fixed-length record; 'VARIABLE' if variable-length record; 'UNKNOWN' if not connected*
SEQUENTIAL = character*	'YES' if sequential access permitted; 'NO' if sequential access is not permitted; 'UNKNOWN' if not connected
UNFORMATTED = character*	'YES' if unformatted records permitted; 'NO' if unformatted records not permitted; 'UNKNOWN' if undetermined

*The specifier variable can be either a variable or array element of the stated type.

7.12.4 File-Positioning Statements

The file-positioning statements allow manipulation of external files. You cannot use these statements with internal files. The positioning statements are:

- REWIND—repositions before the first record
- BACKSPACE—repositions to beginning of preceding record
- ENDFILE—writes an endfile record

The file-positioning statements have the form:

REWIND ([UNIT=]*u* [,ERR=*s*] [,IOSTAT=*ios*])
REWIND *u*

BACKSPACE ([UNIT=]*u* [,ERR=*s*] [,IOSTAT=*ios*])
BACKSPACE *u*

ENDFILE ([UNIT=]*u* [,ERR=*s*] [,IOSTAT=*ios*])
ENDFILE *u*

where:

u is the unit specifier. If the unit specifier is the first argument, you can omit UNIT=.

s is the statement label to which control transfers if an error condition exists. (If both IOSTAT and ERR are omitted, the program terminates on an error.)

ios is an integer variable or integer array element that is set to either a zero if no error condition exists, or a positive integer error code if an error occurs during program execution. (If IOSTAT, without ERR, is included in the statement, execution continues at the next statement on an error.)

7.12.4.1 REWIND Statement

The REWIND statement positions a file at its initial point. If the file is already at its starting point, REWIND takes no action. If the unit is not connected to a file, REWIND has no effect. The following statements reposition the file MYEXAM to its beginning.

```
       ...
    OPEN(10,FILE='MYEXAM',STATUS='OLD')
    READ(10 END=200) A,B,C
       ...
200 REWIND 10
       ...
```

7.12.4.2 BACKSPACE Statement

The BACKSPACE statement positions the file connected to the specified unit before the preceding record. If the file is already at the first record, no action is taken. If the file is positioned after the endfile record, BACKSPACE positions the file before the endfile record. You cannot backspace a file that does not exist.

The following statement repositions the file connected to unit 10 to the beginning of the preceding record.

```
BACKSPACE 10
```

The following statements assign A and B the same value from the file connected to unit 8.

```
READ (8,*) A
BACKSPACE (8)
READ (8,*) B
```

7.12.4.3 ENDFILE Statement

The ENDFILE statement writes an endfile record on the file connected to the specified unit and positions the file after the endfile record. After ENDFILE writes the endfile record, no additional records can be read or written without using BACKSPACE or REWIND to reposition the file for data-transfer operations.

The following statements write endfile records to the files connected to units 101 and 4, respectively.

```
ENDFILE (UNIT=101)
ENDFILE (4)
```

Chapter 8
Format Specifications

Format specifications describe the format of data to be read or written and define any editing that is required. You can use any of the following format specification methods with formatted I/O statements:

- The label of a FORMAT statement that contains the format, for example:

```
      WRITE (6,50) A, B
   50 FORMAT (I4)
```

- An integer variable assigned the label of a FORMAT statement, for example:

```
      ASSIGN 50 TO L
      WRITE (2,L) A1, A2
```

- A character array, character variable, or other character expression that specifies the format, for example:

```
      READ (10,'(I4,I6)') L, M
```

- An asterisk that indicates list-directed input/output, for example:

```
      WRITE (10, *) K, L, M
```

8.1 FORMAT Statement

The nonexecutable FORMAT statement provides editing information necessary to produce the desired format for I/O statements. The FORMAT statement has the form:

sl **FORMAT** (*flist*)

where *sl* is a required statement label and *flist* is a nonempty format list.

Each item in the *flist* is of the form:

[*r*]*ed*   *ned*   [*r*]*fs*

where:

r represents the repeat count.

ed is a repeatable edit descriptor. Repeatable descriptors indicate the type and layout of the next data value in the file. The repeatable descriptors are: I, ***O***, ***Z***, F, E, D, G, A, L, and ***Q***.

ned is a nonrepeatable descriptor. Nonrepeatable descriptors specify format characteristics such as spacing and skipping data that is not required. These descriptors are: H, X, P, T, TL, TR, SP, SS, S BN, BZ, ***B***, ***SU***, ***R***, slash, colon, ***dollar sign***, and apostrophe descriptors.

fs nonempty *flist*.

A repeatable edit descriptor has one of the following forms:

[*r*]*c* [*r*]*cw* [*r*]*cw.m* [*r*]*cw.d*[E*e*] *[r]cw.d.e*

where:

r — is a repeat specification (unsigned integer constant) that indicates repetition of the descriptor *r* times in the format specification. The repeat specification cannot be used with all descriptors. If you omit the repeat specification, the count defaults to 1. You must include at least one repeatable descriptor in the format specification for I/O statements that have one or more items in the I/O list.

c — is a format descriptor that may or may not be repeatable.

w — is an unsigned integer constant that indicates the field width in characters. A field containing only blank characters represents the value of zero. Leading blanks are not significant; other blanks are ignored or represented as zero depending on the value of the BLANK keyword when the file was connected.

m — is an unsigned integer constant that indicates the minimum number of characters, including leading zeros, that must appear within the field.

d — is an unsigned integer constant that indicates the number of characters to the right of the decimal point for REAL values.

E — identifies the exponent field.

e — is an unsigned, integer constant that indicates the number of characters to output as the exponent.

Not all of the previously identified terms are required for formatting. For instance, *e* can be used for formatting real values but is invalid for the use with integer format descriptors, e.g., I, *O*, *Z*. Do not use PARAMETER constants for the terms *r*, *w*, *m*, *d*, or *e*.

8.2 FORMAT Control

When data transfer occurs, format control depends on information provided by the next format descriptor and the next item in the I/O list, if any. Generally, the format specification is interpreted from left to right, and elements in the I/O list are correlated with the corresponding repeatable edit descriptors. There are no corresponding list elements for the nonrepeatable descriptors. The I/O statement terminates if, during execution of the data transfer statement, a repeatable edit descriptor is encountered but there is no corresponding item in the I/O list. For example, the following statement:

```
READ (*, '(I4,5F6.2)' ) K, X, Y
```

causes three values, not six, to be read using the descriptors I4 and F6.2. The additional three F6.2 descriptors are not used. If there is another item in the I/O list but no repeatable descriptor, however, control reverts to the beginning of the format specification, and a new record is started. For example:

```
READ (5, '(I4,I6)' ) K, J, I, N
```

causes values to be read in from character positions 1 to 4, and 5 to 10 of the current record and assigned to K and J, respectively. Control then reverts to the beginning of the format

specification; values from character position 1 to 4, and 5 to 10 of the next record are read and assigned to I and N, respectively. Reversion to the beginning of the format list causes multiple records to be transferred. The end-of-file condition is flagged if there are insufficient records in the file to satisfy the execution of the input statement.

You can transfer data entirely from the descriptors to the external records. In this case, there is no corresponding item in the I/O list for the descriptors so format control communicates information directly to the record. You can use H and character constant descriptors to transfer data directly to the external record from the format specification. For example, the following statement outputs the characters CONVEX FORTRAN to the file connected to unit 2:

```
WRITE (2, ' (''CONVEX FORTRAN'') ')
```

There is no output list in the above WRITE statement. Because the format identifier is a character constant containing the format specification, the apostrophes in the format specification must be represented by two consecutive apostrophes in the format identifier.

Usually, a new I/O statement positions the file at the next record. *The use of $ while writing causes suppression of a new line at the end of the current record.* The slash descriptor terminates processing of the current record; the next record is used for the remaining descriptors.

Processing of repeatable edit descriptors positions the file after the last character transferred. This is also true of the H and apostrophe edit descriptors. Positioning left or right within the current record is accomplished by the X, T, TL, and TR descriptors.

8.2.1 Repeat Count

You can use the descriptors A, *O, Z,* F, E, D, G, L, and I in a repetitive sequence by preceding the descriptor with an unsigned, integer constant that specifies the number of repetitions. For example, the following two statements are equivalent:

```
30 FORMAT (F6.0,F6.0,8X,F10.3,F10.3,F10.3)
30 FORMAT (2F6.0,8X,3F10.3)
```

You can also repeat a group of descriptors by enclosing the descriptors in parentheses and preceding them with an unsigned, integer constant that specifies the number of repetitions. The repeat count defaults to 1 when you do not specify the count. For example, the following two statements are equivalent:

```
30 FORMAT (F6.0,F6.0,8X,F10.3,E12.4,5X,F10.3,E12.4,5X,F4.0)
30 FORMAT (2F6.0,8X,2(F10.3,E12.4,5X),F4.0)
```

8.3 Descriptors

The field and edit descriptors in CONVEX FORTRAN are grouped into the following categories:

- Character (A)
- Editing, character constants, and Hollerith constants (T, TL, TR, P, *Q, dollar sign,* colon, ..., slash, X, H, *B,* BN, BZ, S, SP, SS, *SU, R*)
- Integer (I, *O, Z*)
- Logical (L)
- Real and complex (D, E, F, G)

8.3.1 A Descriptor

The A descriptor transfers character or Hollerith values and is represented by:

A[*w*]

In an input statement, the A field descriptor transfers *w* characters from the external record and assigns them to the corresponding I/O list element. If the *w* field is not specified, the size is the length of the character variable, character substring reference, or character array element. For numeric I/O list elements, the size depends on the data type, as shown in Table 8-1.

Table 8-1: Character Assignment for Numeric I/O List Elements

I/O List Elements	Maximum No. of Characters
*LOGICAL*1*	*1*
*LOGICAL*2*	*2*
LOGICAL*4	4
*LOGICAL*8*	*8*
*INTEGER*1*	*1*
*INTEGER*2*	*2*
INTEGER*4	4
*INTEGER*8*	*8*
REAL*4	4
REAL*8(DOUBLE PRECISION)	8
COMPLEX	8
*COMPLEX*16(DOUBLE COMPLEX)*	*16*

If the *w* you specify is less than the size of the *iolist* item, on input the characters are stored left justified and padded on the right with blanks. If the *w* you specify is greater than the size of the *iolist* item, the rightmost characters are stored in the variable on input.

On output, the value is right justified in the field and *w* characters from the entity are written to the record. If *w* is greater than the number of characters in the entity, leading blanks are added to right justify the value. It *w* is less than the number of characters in the *iolist* item, only the leftmost *w* characters are written.

Examples:

Reading into a CHARACTER*5 variable:

Format Code	External Field	Internal Value
A	CONVEXCOMPUTER	CONVE
A4	CONVEXCOMPUTER	CONV^
A14	CONVEXCOMPUTER	PUTER

Writing from a CHARACTER*10 variable:

Format Code	Internal Value	External Field
A	MY^EXAMPLE	MY^EXAMPLE
A4	MY^EXAMPLE	MY^E
A14	MY^EXAMPLE	^^^^MY^EXAMPLE

8.3.2 Apostrophe Descriptor

The apostrophe descriptor has the form of a character constant. The characters that are enclosed within a pair of apostrophes are written to the record. The width of the field equals the number of characters contained within (but not including) the delimiting apostrophes. Use two consecutive apostrophes to produce a single apostrophe. For example, the statement:

```
      WRITE (6,100)
100 FORMAT ('THE^''CONVEX''^COMPUTER')
```

produces

```
THE 'CONVEX' COMPUTER.
```

8.3.3 H Descriptor

The H descriptor writes a literal string to a record. You can use the H descriptor for output editing as an alternative to apostrophe editing. This descriptor has the form:

*n***H***c...c*

The H descriptor writes the *n* characters immediately following the letter H, including apostrophes and quotation marks. The *c...c* represents the actual characters to be written. For example, the statement:

```
    WRITE (6,10)
10 FORMAT (17HENTER 'FILE' NAME)
```

produces:

```
ENTER 'FILE' NAME
```

8.3.4 L Descriptor

The L descriptor formats logical variables and has the form:

L*w*

where *w* indicates the field width for the formatting of logical variables.

Optional blanks, optionally followed by a decimal point, a T (t, .T, .t) for true or an F (f, .F, .f) for false, constitute the input field. The T or F can be followed by additional characters, i.e., .TRUE. or .FALSE., in the field. On input, an all blank field is read as *false*.

On output, the record contains *w* - 1 blanks, followed by a T or F depending on the value of the corresponding I/O list element.

Input Example:

Format Code	External Field	Internal Value
L2	T60	.TRUE.
L7	^^FALSE	.FALSE.
L7	1234567	Error—invalid

Output Example:

Format Code	Internal Field	External Field
L1	.TRUE.	T
L3	.FALSE.	^^F

8.3.5 I Descriptor

The I descriptor provides integer formatting. It has one of the forms:

I*w* or I*w.m*

where:

w is an unsigned, positive integer constant that specifies the field to be edited is *w* characters in width.

m is an unsigned, integer constant that specifies the minimum number of digits for output (only), including leading zeros if necessary.

During input, the processor transfers *w* characters from the record in integer representation and stores the integer values in the corresponding I/O list elements. Both forms of the I descriptor are treated identically during input. On input, leading blanks are not significant; nonleading blanks are interpreted according to the BLANK specifier in the OPEN statement or the BZ or BN descriptor. If the field contains all blanks, the value is zero. A plus sign or no sign indicates a positive value; a leading minus sign indicates a negative value.

During output, the processor formats the value of the I/O list element and outputs it in a field *w* characters wide, right justified. Leading blanks are added, if needed, to fill the field. If the field specified is too small for the value, the field fills with asterisks (*). If you specify *m*, the external field consists of, at most, *m* characters; if necessary, the processor inserts leading zeros to pad to *m*. The value of *m* must not be greater than the value of *w*. If *m* is zero and the value of the entity is zero, the output field contains blank characters. The minus sign precedes a negative integer; by default a plus does not precede a positive integer. You must include space for a minus sign for negative integers in the *w* term.

Input Examples:

Format Code	External Field	Internal Value	
I3	760	760	
I4	^^^^	0	
I4	-760	-760	
I5	760^^	760	
I5	760^^	76000	(blanks interpreted as zeros)
I5	7.60^^	Error	

Output Examples:

Format Code	Internal Value	External Field
I4	760	^760
I8.4	760	^^^^0760
I3	-760	***
I4	0	^^^0

8.3.6 O Descriptor

The O descriptor transfers unsigned octal values. The descriptor has the form:

Ow[.m]

where:

w is an unsigned, positive integer constant that specifies the field to be edited is *w* characters in width.

m is an unsigned, integer constant that specifies the minimum number of digits for output, including leading zeros if necessary.

On input, format code O transfers w characters from the external field and assigns them as an octal value to the corresponding I/O list element. On output, if m is specified and the external field consists of fewer digits than m, the remaining positions fill with zeros on the left. You can use only the numerals 0 - 7 in the external field; you cannot use a decimal point, a sign, or an exponent field.

Input Examples:

Format Code	External Field	Internal Octal
O3	523	523
O4	23176	2317
O4	2.317	Error
O4	-1234	Error

Output Examples:

Format Code	Internal Decimal Value	External Value
O6	4095	^^7777
O6	-4095	*****
O3	4095	***
O4.3	8	^010

8.3.7 Z Descriptor

The Z descriptor transfers unsigned hexadecimal values and is represented as:

Zw[.m]

where:

w is an unsigned, positive integer constant that specifies the field to be edited is *w* characters in width.

m *is an unsigned, integer constant that specifies the minimum number of digits for output, including leading zeros if necessary.*

On input, descriptor Z transfers w characters from the external field and assigns them as a hexadecimal value to the corresponding I/O list element. On output, if m is specified and the value contains fewer digits than m, the remaining positions fill with zeros on the left. You can use only the numerals 0 - 9 and the letters A (a) through F (f) in the external field; you cannot use a decimal point, a sign, or an exponent field.

Input Examples:

Format Code	*External Field*	*Internal Hex Value*
Z3	*9A1*	*9A1*
Z3	*9A1B*	*9A1*
Z3	*9A.1*	*Error*

Output Examples:

Format Code	*Internal Decimal Value*	*External Value*
Z4	*4095*	*^fff*
Z5	*-1*	*******
Z6.4	*4095*	*^^0fff*
Z2	*4096*	****

8.3.8 F Descriptor

The F descriptor provides formatting of real numbers. It has the form:

F*w.d*

where:

w is an unsigned, positive integer constant that specifies the field to be edited is *w* characters in width.

d specifies the number of digits in the fractional (right of the decimal) part of the real number.

During input, the processor transfers *w* characters from the external field and stores the real values in the corresponding I/O list elements. The input field consists of an optional sign followed by a string of digits that can contain a decimal point. If the field contains a decimal point, the *d* term has no effect, as the location of the explicit decimal overrides the location specified by the field descriptor. If you omit the decimal point and the exponent, the rightmost *d* digits are interpreted as the fractional part of the field with leading zeros assumed if necessary.

On input, leading blanks are not significant; nonleading blanks are interpreted according to the BLANK specifier or the BZ or BN descriptor. If the field contains all blanks, the value is zero. The processor treats a plus sign or no sign as a positive value; the minus sign indicates a negative value.

During output, the processor transfers the value of the I/O list element rounded to *d* decimal positions and outputs it in a field *w* characters wide, right justified. *w* must include space for a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal, i.e., at least equal to or greater than $d + 3$. Leading spaces are added, if needed, to fill the field.

Input Example:

Format Code	External Field	Internal Value
F8.5	1234567^	12.34567
F8.5	12345.67	12345.67
F8.0	-1.23E-3	-.00123
F8.5	123456789	123.4568

Output Example:

Format Code	Internal Value	External Field
F9.4	123.456789	^123.4568
F5.2	123.456789	*****
F6.3	+1.12	^1.120
F6.3	-1.12	-1.120

If the value is too large for the field, asterisks (*) are output. In native format, if the sign is a 1 and the exponent is 0, *Rop* (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, *NaN* (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, *Inf* (infinity) is output.

8.3.9 E and D Descriptors

The E and D descriptors are functionally identical. Both transfer real values in exponential form and edit external real, double-precision or complex data. These descriptors differ only in the exponent symbol they use. The E and D descriptors have the following forms:

E*w.d*, *Ew.d.e*, or E*w.d*E*e*
D*w.d*, *Dw.d.e*, D*w.d*E*e*

where:

w is the width of the field containing the real number, i.e., count of all characters in the field, including sign, if any, decimal point, and exponent.

d is the fractional part of the field that consists of *d* digits.

E or D identifies the exponent part that consists of *e* digits (no effect on input).

e indicates the number of digits in the exponent.

On input, the descriptors read *w* characters from the external field and assign them as a real value to the corresponding I/O list element. The values being read consist of a string of digits with an optional decimal point. When the decimal point is included, the *d* term has no effect. When the decimal is omitted, however, the least significant *d* digits of the string are considered the fractional part of the value.

On output, the E and D descriptors transfer the value of the I/O list element rounded to *d* decimal positions and output it to a field *w* characters wide, right justified. *w* must include space for a minus sign (when necessary), the decimal point, *d* digits to the right of the decimal, and a two- or three-digit exponent, depending on whether the I/O list item is REAL*4 or REAL*8.

The exponent can be a signed-integer constant, (E or D) followed by zero or more blanks, followed by an optional signed-integer constant. You can use the legal characters—digits 0 - 9, decimal point, plus, minus, E, D, and blank. When using the descriptors in the form of *Ew.d*, ***Ew.d.e*** or *Ew.dEe* (*Dw.d, Dw.d.e*, ***Dw.dEe***), the value of the next item in the output list has the form:

$[\pm][0].x_1,x_2 \dots x_d \; exp$

where:

±	signifies a plus or minus sign; the plus is optional for a positive value.
0	a leading zero that is optional.
$x_1,x_2 \dots x_d$	are the *d* most significant digits of the value after rounding.
exp	is a decimal exponent that is of the form $E\pm z_1z_2z_3$ where *z* is a digit.

Input Example:

Format Code	External Field	Internal Value
E4.3	.625	.625
E6.1	.78-01	.078
D6.3	-.62D4	-6200
D6.3	123456	123.456
E4.3	62567	6.256

Output Example:

Format Code	Internal Value	External Field
D11.4	-6250.	-0.6250D+04
E10.3	625	^0.625E+03
E10.4	0.4568	0.4568E+00
E10.3	0.4568	^0.457E+00
E5.3	24.53	*****

If the value is too large for the field, asterisks (*) are output. In native format, if the sign is a 1 and the exponent is 0, *Rop* (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, *NaN* (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, *Inf* (infinity) is output.

8.3.10 G Descriptor

The G descriptor edits external single-precision, double-precision, or complex data. It has the form:

G*w.d*, *Gw.d.e* or G*w.d*E*e*

where:

w	is a nonzero, unsigned, integer constant that indicates the field width in characters.
d	is a nonzero, unsigned, integer constant that indicates the number of characters to the right of the decimal point.

E identifies the exponent field.

e is an unsigned, integer constant that indicates the number of digits in the exponent.

Input editing is identical to F, E, and D editing. You may use the G descriptor when you are not certain that the values you are using can be adequately represented by the F descriptor because of their magnitude—either too large or too small.

On output, the G descriptor operates like the F descriptor. It transfers the corresponding I/O list item rounded to *d* digits. Either the F or E style of editing is used depending on the magnitude of the value. If the value can be represented using the F format without loss of significant digits, F is chosen; otherwise, E is chosen.

Assume M is the magnitude of the data in the field. If M is less than 0.1 or equal to or greater than 10**d, the output editing Gw.d or Gw.dEe is the same as kPEw.d and kPEw.dEe, respectively, and *k* is the scale factor currently in effect. If M is equal to or greater than 0.1 or less than 10**d, however, the F mode of editing is used with output of the four-character exponent field as four blanks after the value. The scale factor has no effect and the value of M determines the editing as follows:

Magnitude of Data	Conversion Equivalence
0.1≤M<1.0	F(w-n).d,n(' ')
1.0≤M or <10.0	F(w-n).(d-1),n(' ')
.	.
.	.
.	.
10**(d-2)≤M<10**(d-1)	F(w-n).1,n(' ')
10**(d-1)≤M<10**d	F(w-n).0,n(' ')

The value *n(' ')* specifies that four or *e* + 2 spaces are to follow the numeric data representation; *n* is 4 for Gw.d and e + 2 for Gw.dEe. Be sure the *w* term is large enough to include a sign, if necessary, a decimal point, *d* digits to the right of the decimal and either a 4-character or *e* + 2-character exponent. Thus, you must make *w* equal to or greater than *d* + 7 or *d* + 5 + *e*.

Input Example:

Format Code	External Field	Internal Value
G8.5	^1234567	12.34567
G8.5	12345.67	12345.67
G8.0	-1.234-3	.001234

Output Example:

Format Code	Internal Value	External Field
G13.6	-1234	^-1234.00^^^^
G13.6	0.01234	^0.123400E-01
G13.6	1.23456789	^^1.23457^^^^
G10.4	15.65	^15.65^^^^
E10.4	15.65	0.1565E+02
F10.4	15.65	^^^15.6500

If the value is too large for the field, asterisks (*) are output. In native format, if the sign is a 1 and the exponent is 0, *Rop* (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, *NaN* (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, *Inf* (infinity) is output.

8.3.11 B Descriptors

The B descriptors operate only during execution of the input statements and affect only the numeric descriptors I, *O*, *Z*, F, E, D, and G. The BN and BZ descriptors supercede the default interpretation of blanks while the ***B descriptor causes return to the default mode of blank interpretation***. Their forms and meanings are:

B ***reverts to default interpretation.***

BZ interprets blanks as zeros.

BN interprets blanks as nulls.

When execution of a formatted input statement begins, the interpretation of blanks depends on the value of the BLANK specifier on the OPEN statement, i.e., ignore or zero. Blanks are ignored ('NULL') if you omit the BLANK specifier.

The BN descriptor causes the processor to treat all embedded blank characters as nulls in subsequent input fields for the current statement. When the processor encounters BN, it treats the input field as though the embedded blanks have been moved to the position of leading blanks; the remainder of the field becomes right justified. The processor assigns the value of zero to a field of all blanks. If you specify the BZ descriptor, the processor treats all embedded and trailing blanks in subsequent numeric input fields as zeros.

For example, if a file connected to unit 5 contains the record:

```
^^^5^-500^^^3^056
```

and the BLANK specifier has a value of 'NULL' or the BN descriptor is specified, the statements:

```
READ (5,'(I4,I7,I6)') L, M, N
READ (5,'(BN,I5,I7,I6)') L, M, N
```

assign the value of 5 to L, the value of -500 to M, the value of 3056 to N. The processor ignores all blanks. If the BZ descriptor is indicated by

```
READ (5,'(BZ,I5,I7,I6)') L, M, N
```

the values assigned become: L = 50; M = -50000; N = 30056. The processor treats all nonleading blanks as zero. If another input statement refers to unit 5, blank interpretation returns to the default value.

The descriptor B causes return to the default mode of blank interpretation ('NULL') and is identical to BN. For example, change the previous example to include a B descriptor, such as:

```
READ (5,'(BZ,I5,I7,B,I6)') L, M, N
```

The value of 3056 is assigned to N rather than 30056 as the B descriptor returns interpretation to default mode.

8.3.12 P Descriptor

The P descriptor specifies a scale factor for real and complex values. The P descriptor can be used on input or output and applies to the F, E , D, and G edit descriptors. The P descriptor has the form:

*n*P

where *n* is an optionally-signed integer constant that specifies the number of positions, to the left or right, that the decimal point is to be moved. A value for *n* must be specified.

If no P descriptor is specified, a scale factor of 0 is assumed. Once specified, a scale factor remains in effect within a FORMAT statement until another P descriptor is encountered. The following example uses a scale factor of 0 for the first format descriptor and a scale factor of 2 for the two remaining descriptors.

```
   PRINT 50, D
50 FORMAT (5,'(F8.2,2PF8.2,F6.2)')
```

On input, the scale factor (with the F, E, D, or G descriptors) causes the externally represented number to be multiplied by 10**-*n* before it is assigned to the I/O list element.

Examples:

Format Code	External Field	Internal Value
3PF7.4	56.789^	.056789
-3PE6.3	56.789	56789.

On output, when you use the scale factor with the F descriptor, the externally represented number equals the internally represented number multiplied by 10***n*. When the scale factor is used with E or D, the nonexponent part of the constant is multiplied by 10***n* and *n* is subtracted from the exponent. With G, if the F style of formatting is used, the scale factor is ignored; otherwise, the effect is the same as E editing.

Examples:

Format Code	Internal Value	External Field
-1P,F7.3	58.967	^^5.897
2PE10.3	890.11	^89.01E+01

If you use a scale factor when an external field contains an explicit exponent, e.g., 5.E02, the processor ignores it; in this case, 500 is stored regardless of the scale factor.

8.3.13 S Descriptors

The S descriptor can be used to control optional plus (+) characters in numeric output *or to cause integer values to be interpreted as unsigned during output conversion.* If you do not use any S descriptor, positive values do not have leading plus signs. The S, SP and SS descriptors act only during statement execution and only with I, F, E, and D editing. *The SU descriptor only affects integer values.* The descriptors and their forms are:

S reverts to normal interpretation

SP adds plus sign

SS suppresses plus sign

SU outputs integer values as unsigned values

The SP descriptor forces a plus sign during output for all subsequent positive I, F, D, E, and G values within the format specification. Include space for the plus sign in the numeric fields. When you use the SS descriptor, the processor suppresses leading plus characters from any position where the plus is normally contained as an optional plus. The S descriptor returns the normal plus sign handling option to the processor. For example, if L = +5, M = 100, N = -10, I = 50, J = 6000, and K = -450, the following statements:

```
    WRITE (10,30) L, M, N, I, J, K
30 FORMAT (SS,I2,I5,SP,I4,I4,S,I5,I5)
```

write the record as

```
^5^^100^-10^+50^6000^-450.
```

The SU descriptor, causes integer values to be interpreted as unsigned during output conversion. SU remains in effect until another sign control specifier is encountered or until format interpretation is complete. It has no effect on input. Radix and unsigned specifiers can be used to format a hexadecimal dump as follows:

2000 FORMAT (SU, 16R, 8I10.8)

8.3.14 R Descriptor

The R descriptor changes the radix for integer I/O. This descriptor applies only to integers (I descriptor) and must not be used with other descriptors. The R specifier has the form:

[n]R

where 2 ≤ n ≤ 36. The default value is 10. Omitting n restores the default decimal radix. The radix specified by R remains in effect until another radix is specified or until format interpretation is complete.

For example,

I = 15
WRITE (6,10) I, I, I 10 FORMAT (16R,I4,8R,I4,R,I4)

produces ^^^F^^17^^15.

8.3.15 X Descriptor

The X descriptor sets the position in a record and has the form:

*n*X

where *n* indicates the number of character positions to move forward (skip over) from the current position in the file. The value of *n* must be greater than or equal to 1. The default is 1.

The X descriptor is functionally identical to the TR descriptor. When you use the X descriptor, *n* indicates the next *n* characters are to be skipped. The character following the number of skipped positions is transmitted. For example, the statements:

```
WRITE (*,200) 450, 8921
```

```
200 FORMAT (2X,I3,3X,I4)
```

insert 2 blanks before 450 and 3 blanks before 8921.

The X format descriptor may not in itself change the length of a record. For example, the following is invalid:

```
    WRITE (6,100) I
100 FORMAT (I4,X)
```

8.3.16 T Descriptors

The T (tab) descriptors control forward and backward positioning within a record for input or output of characters. These descriptors let you skip portions of a record or reread portions of a record. The T descriptors are T, TR, and TL.

The T descriptor has two forms; the first form is:

T*n*

where *n* specifies the absolute position within the record. It indicates transmission of characters at position *n*. For example, in a file connected to the designated input unit is the record:

```
^2.5^200^^40
```

Execution of the following statements:

```
   READ (*,35) A, B
35 FORMAT (T2,F3.0,T11,F3.0)
```

assigns A the value of 2.5 (positions the file at character 2 and reads the next 3 characters according to format specification F3.0) then assigns the value of 40 to B (positions the file at character 11 of the record and reads the next 3 characters).

On output, for example, the following statements:

```
   PRINT 50
50 FORMAT (T10,'MY',T13,'EXAMPLE')
```

output MY at position 10 and EXAMPLE outputs at position 13.

Another form of the T descriptor is:

T or nT

which causes tabbing to the next (or n-th) 8-column tab stop. You can therefore align columns of alphanumerics without counting. For example, the statements:

```
   READ (5,50) K,N
50 FORMAT (T,I4,2T,I3)
```

causes K to be read starting in character position 8 of the current record; the value for N is read starting in position 24 of the current record.

The second of the T-series descriptors has the form:

TL*n*

where *n*, an unsigned, integer constant, indicates that the record is repositioned *n* characters left (backwards) from the current position in the record. The default is 0. For example, if the external record is 1.2345, the statements:

```
   READ (5,20) X, I
20 FORMAT (F6.0,TL4,I3)
```

produce: X = 1.2345 and I = 234

The last T-series descriptor is TR; it has the form:

TR*n*

where *n* is an unsigned integer that specifies the number of characters to move right (forward) from the current position in the record. The default is 0. The TR field descriptor is identical to X. For example, assume the external record is:

```
12.345^^^^123
```

the statements:

```
   READ (5,20) X, I
20 FORMAT (F6.0,TR4,I3)
```

produce: X = 12.345 and I = 123

The T descriptors cannot in themselves change the record length. Therefore, the following results in a runtime error:

```
   WRITE (6,10) I
10 FORMAT (I4,TR10)
```

8.3.17 $ Descriptor

The $ descriptor suppresses the new line at the end of the current record of a formatted sequential write. (In an input statement, the $ descriptor is ignored.) For terminal I/O, a typed response follows the output on the same line. For example, the following statements:

```
WRITE (*,'(" enter value for x: ",$)')
READ (*,*) x
```

write ^enter value for x:^ to the output device with the cursor positioned one space to the right of the colon.

8.3.18 Q Descriptor

The Q descriptor determines the number of unread characters in the current record. It is represented by:

Q

For example:

```
    READ (4,100) J, MYEXAM, (ISAM(I), I = 1, MYEXAM)
100 FORMAT (I5,Q,80AI)
```

reads the first field into variable J and stores the number of remaining characters in MYEXAM and causes transfer of that number of characters to the character array, ISAM. If you place Q first in the format specification, you can determine the actual length of the record.

In an output statement, the descriptor Q causes the corresponding I/O list element to be skipped.

8.3.19 Colon Descriptor

The colon (:) descriptor terminates format control when there are no more items in the I/O list. If items remain in the I/O list, the colon descriptor has no effect. For example:

```
      M = 15
      WRITE (10,40)M
   40 FORMAT (I2,:,' SAMPLE',I3)
```

writes 15 only, ending format control at the colon. Change the statements slightly, however, and the colon descriptor has no effect. For example:

```
      M = 15
      N = 500
      WRITE (10,40) M, N
   40 FORMAT (I2,:,' SAMPLE',I4)
```

writes 15 SAMPLE 500; the colon descriptor is ignored as items remain in the I/O list.

8.3.20 Slash Descriptor

The slash descriptor indicates the end of data transfer for the current record. For example, the following statements:

```
      READ (10,50) L, M, N
   50 FORMAT (I2/I4,I3)
```

cause L to be read from the first record; M and N, from the second record.

During input, use sequential slashes to indicate bypassing of records. The first slash indicates the end of input for the current record; subsequent slashes skip records. When you use the slash on a unit connected for sequential access, the remainder of the current record is skipped and the file is positioned at the beginning of next record. On direct access, 1 is added to the record number and the processor reads that record.

On output, slashes can be used to create empty records. The first slash indicates end of output for the current record; subsequent slashes produce empty records.

8.4 Default Field Descriptor Values

If you do not specify a field width value with the field descriptors I, *O*, *Z*, L, F, E, D, G, or A, default values for *w*, *d*, and *e* are supplied based on the data type of the I/O list element as shown in Table 8-2.

Table 8-2: Default Field Descriptors

Field Descriptor	List Element Type	w	d	e
I,O,Z	INTEGER*1,LOGICAL*1	7		
I,O,Z	INTEGER*2,LOGICAL*2	7		
I,O,Z	INTEGER*4,LOGICAL*4	12		
I,O,Z	INTEGER*8,LOGICAL*8	23		
O,Z	REAL*4	12		
O,Z	REAL*8	23		
L	LOGICAL	2		
F,E,G,D	REAL,COMPLEX*8	15	7	2
F,E,G,D	REAL*8,COMPLEX*16	24	15	3
A	LOGICAL*1,INTEGER*1	1		
A	LOGICAL*2,INTEGER*2	2		
A	LOGICAL*4,INTEGER*4	4		
A	LOGICAL*8,INTEGER*8	8		
A	REAL*4,COMPLEX*8	4		
A	REAL*8,COMPLEX*16	8		
A	CHARACTER*N	n		

8.5 Comma Field Separator on Input Data

A comma between numeric fields overrides the width specified in the field descriptor. Because you can use a comma to terminate a field, you can avoid padding the input field, which makes entering data from a terminal keyboard easier. A comma field separator can be used with the numeric descriptors (I, O, Z, F, E, D, G, and L).

Example:

```
     READ (5,100) I,K
100  FORMAT (2I4)
```

Record:

```
1,2
```

Result:

```
I = 1
K = 2
```

The following constraints apply:

- *Two successive commas constitute a null field.*
- *You cannot use a comma to terminate a field that is controlled by an A, or character constant field descriptor. If the record reaches its physical end before w characters are read, short-field termination occurs and the characters you input are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list item.*

8.6 Runtime Formats

Format specifications, called runtime formats, can be stored in character variables, character substrings, character expressions, and in character and *numeric arrays and numeric array elements. The numeric arrays and numeric array elements are treated as Hollerith constants.*

You can define or modify a runtime format during program execution. A runtime format is similar to a FORMAT statement but does not have a label or the word FORMAT. For example:

```
INTEGER*8
CHARACTER*8 SFMT
IFMT = 8H(2X,I12)
SFMT = '(2X,I12)'
WRITE (6,IFMT) I
WRITE (6,SFMT) I
```

8.7 Variable Formats

A variable format contains an expression, enclosed in angle brackets, that is recomputed each time it is encountered during format scanning. The expression has the form:

<*expression*>

The expression in the angle brackets can be used in a FORMAT statement wherever you can use an integer, except as the character count of a Hollerith (H) descriptor.

A variable expression in a format is subject to the following rules:

- If the expression is not integer, it is converted to integer before use.
- Any valid FORTRAN expression can be used, including function calls and dummy argument references.
- The value of the expression must conform to the restrictions on magnitude that apply to its use in a format.
- A variable expression is not allowed in a runtime format.

Do not perform I/O operations within a function call used in a variable format expression or a runtime error occurs.

Example:

```
C test of d and e descriptors with repeat count
    1 format(<j+2>D10.4,<j/2+1>E10.4)
    2 format(<j+2>D10.4.2,<j/2+1>E10.4.2)
      j = 2
      read(5,1) a,b,c,d,e,f
      write(6,2) a,b,c,d,e,f
      stop
      end
```

8.8 List-Directed Formatting

List-directed formatting transfers data based on the data type of the entity. A list-directed I/O statement contains an asterisk (*) as the format indicator. For example, the following statement:

```
READ (5,*) J, M, L
```

reads three fields from unit 5 and assigns integer values to the variables J, M, and L.

The list-directed record consists of a sequence of values and value separators. A value is generally a constant but can also be a null value, or the value may have the forms:

*r**c or *r**

where:

r is an unsigned, nonzero, integer constant that represents the repeat count.

*r*c* represents successive appearances of the constant *c*. You can enter a repeat count to assign a value to more than one entity with *r*c*.

*r** repeat count with an empty constant (null values). A null value indicates that the value of the corresponding I/O entity is to remain unchanged.

Separators divide the values in each list-directed record. A value separator is a blank, comma, or a slash optionally enclosed by blanks. Normally, the blanks are considered as part of some value separator. In the following cases, the blanks are not considered part of a value separator:

- Leading blanks in the first record unless followed by a slash or comma.
- Blanks embedded in a character constant.

8.8.1 Input

You can use list-directed input from any file that allows formatted input. The data type of the constant, which can be logical, integer, real, complex, or character, determines the data type of the value, as well as the translation from external to internal form. A character list element must correspond with a character constant; likewise, a numeric element with a numeric constant. If the data type of the external numeric field does not match the data type of the numeric list item, the external value is converted according to the rules for conversion on assignment (see Table 5-1). Input fields are separated by blanks, commas, or slashes.

The format of a complex value is: left parenthesis followed by a numeric value, a comma, another numeric value (an ordered pair of numeric fields separated by a comma), followed by a right parenthesis. The processor ignores one or more blanks around either parenthesis or the comma. The end of record can occur between the real part and the comma or between the comma and the imaginary part.

8.8.1.1 Character Input

Character constants for list-directed input are usually enclosed in apostrophes. Character constants may span record boundaries.

Embedded blanks, commas, and slashes are not considered separators within a character string. To include an apostrophe as part of a character string, use two consecutive apostrophes without an intervening blank or end of record.

The processor transfers the leftmost characters read, either truncating the constant to fit in the list item or filling it on the right with blanks.

CONVEX FORTRAN allows input of a string not enclosed in quotes. The string must not start with a digit and cannot contain a separator consisting of a right or left parenthesis, i.e., (' or '), or blank (space or tab). A "newline" terminates the string unless escaped with "\". Any string not meeting these restrictions must be enclosed in single or double quotes.

8.8.1.2 Nulls and Slashes

You can specify a null value for a list item with a comma or with $r*$ in the external record. No characters between successive value separators or no characters preceding the first value separator indicate a null field. When assigning a null for the first value, you can use one comma; for a subsequent null, use two consecutive commas.

When you use a null value, it does not alter the value of the corresponding input list item.

When the processor encounters a slash on list-directed input, it skips the rest of the I/O list items and terminates the READ statement. Those items skipped retain their original values.

8.8.2 Namelist-Directed Formatting

To assign input values for a namelist-directed READ, you must delimit the input record (or records) with a dollar sign ($). Namelist input has the following form:

$nlgrpname [ent = value [,]] ... $[END]

where:

$	*is the special symbol that indicates the beginning and end of input. You can use the ampersand (&) rather than the $.*
nlgrpname	*is the name defined for the entities contained in the namelist.*
ent	*is a namelist entity. The entity can be a variable, an array name, a subscripted variable, a variable with a substring, or a subscripted variable with a substring.*
value	*is a constant, a list of constants, or a repetition of constants or null values.*
END	*is an optional delimiter indicating no more input.*

Use constant values for assigned values, array subscripts, and substring specifiers; you cannot use PARAMETER constants.

You can use any data type. Conversion (following rules of arithmetic assignment) is performed if the data type of a namelist entity and its assigned constant value do not match. Conversion between numeric and character data is not allowed.

The following example shows how to input data to the namelist entities. You can assign the values in any order.

```
$SAM
NAME = 'TESTA',
EXAM1 = 5.2,
```

```
EXAM2 =  6.78,
EXAM3 = 10.0
$
```

There are several acceptable formats for entering input. For example, the previous input could also be entered:

```
$SAM^NAME^=^'TESTA'^EXAM1^=^5.2^EXAM2^=^6.78^EXAM3^=^10.0 $END
```

You can also use commas or tabs as valid separators in the list of value assignments, as well as spaces. (Input can begin at any column.)

The previous example assigns values to all of the namelist entities associated with SAM; however, you do not need to assign values to all the defined entities. Only those entities that you assign a value to change; those defined in the namelist but not assigned a value in the input data remain unchanged. Likewise, when you have defined character substrings and array elements in the namelist, only those you specify to receive input are changed. You can change part of a character substring. For example, to change the character variable NAME from 'TESTA' to 'TESTB', use the following namelist-directed input:

```
$SAM NAME(5:) = 'B' $END
```

The value for NAME is 'TESTB'; the first four positions of the value remain unchanged.

When you assign values to an array name, the first value is associated with the first element; the second value with the second element, etc. The size of the array determines the number of array elements you can assign; assignment must be less than or equal to the size of the array.

Using a program that contains:

```
DIMENSION MYRAY(10)
NAMELIST /SAM2/ MYRAY
READ SAM2
```

with the following input:

```
$SAM2 MYRAY = 10, 8, , 70 $END
```

Then, on program execution, the READ statement assigns the following values to the array elements:

```
MYRAY(1)         10
MYRAY(2)          8
MYRAY(3)      Unchanged
MYRAY(4)         70
MYRAY(5-10)   Unchanged
```

Values MYRAY (3) and (5-10) remained unchanged since two consecutive commas indicate to not change the current value, and values for unspecified arrays also remain unchanged.

Because values are assigned to the specified array element rather than beginning with the first element, the READ statement can assign new values and not alter unspecified elements. For example, the following line assigns values to MYRAY elements 5-7; the unspecified elements remain unchanged:

```
$SAM2 MYRAY(5) = 9, 85, 60 $END
```

Namelist-directed formatting follows the rules for list-directed input:

- *When you define a groupname, do not use spaces or tabs in the name. Also, in value assignments, the entity name cannot contain spaces or tabs except within a subscript or substring specifier. Then spaces and tabs are acceptable within the parentheses.*
- *The groupname and each entity must be contained within a single record.*
- *When assigning values, you can precede and follow the equals sign with any number of tabs and spaces.*
- *Character constants are enclosed in apostrophes. If you want an apostrophe to appear as part of the character string, use two consecutive apostrophes without an intervening blank or end record.*
- *You cannot use Hollerith, octal, or hexadecimal constants.*
- *Character constants may span record boundaries. Normally, the end of a record in namelist input is a space character. If the end of record occurs within a character constant, however, the end of record is ignored; the last character of the previous record is followed by the first character of the next record.*
- *For fixed record length files, NAMELIST produces records of a fixed length, and reads records of a fixed length for that type of file.*

8.8.3 List-Directed Output

The format of list-directed output is defined by the data type of the I/O list items except that r^* is not used. Also, quotation marks are not output for character constants. Table 8-3 shows the default output forms that the list-directed WRITE statement generates for each data type.

Table 8-3: List-Directed Output Formats

Data Type	Output Format
*LOGICAL*1*	*L2*
*LOGICAL*2*	*L2*
LOGICAL*4	L2
*LOGICAL*8*	*L2*
*INTEGER*1*	*I5*
*INTEGER*2*	*I7*
INTEGER*4	I12
*INTEGER*8*	*I22*
REAL	1PG15.7E2
REAL*8	1PG24.15E3
COMPLEX	1X,'(',1PG14.7E2,',',1PG14.7E2,')'
*COMPLEX*16*	*1X,'(',1PG23.15E3,',',1PG23.15E3,')'*
CHARACTER	An (n represents the character expression length)

8.8.4 Namelist-Directed Output

The format of namelist-directed output is defined by the data type of the list entities in the corresponding NAMELIST statement. When you use a namelist-directed WRITE statement, the order of data output is specified by the sequence in which namelist entities are defined in the NAMELIST statement. For example, if you have a program unit that contains:

```
LOGICAL L4
INTEGER I4
REAL R4
COMPLEX C8
CHARACTER*20 CHAR20

NAMELIST /CONTROL/ L4,I4,R4,C8,CHAR20

READ (5,CONTROL)
WRITE (6,CONTROL)

END
```

with the following input:

```
$CONTROL
                L4 = F, I4 = -123213, C8 = (12,2),
                CHAR20='test case',
                R4=3.14159
$END
```

the WRITE statement outputs:

```
$CONTROL
L4      = F,
I4      = ^^^^-123213,
R4      = ^^3.141590^^^^,
C8      = (^^12.00000^^^^,^^2.000000^^^^),
CHAR20  = 'test^case^^^^^^^^^^^'
$END
```

The output for this program segment is the current values of all list entities associated with the namelist specifier. You can output a value for an entity that is defined by the NAMELIST statement but is assigned no input value. (The entity may also be undefined or defined elsewhere in the program.) For instance, if you had an entity, count, that was defined in the NAMELIST statement but received no input, the current value of count would be written in addition to those values shown in the example.

As the example illustrates, each value begins on a new line for namelist-directed output. Character values are enclosed in apostrophes. As stated previously, the data types are determined by the data type of the entities defined in the NAMELIST statement. The format output follows the same form as list-directed. Although you can use the $ and & characters interchangeably on input, the $ character is always used for output.

8.9 Carriage-Control Characters

The first character of a formatted record transfers to the printer as a carriage-control character. Table 8-4 shows the characters that provide vertical format control.

Table 8-4: Vertical Format Control

Character	Interpretation
^	Advance 1 line; begins output at beginning of next line.
0	Advance 2 lines; skips 1 line and begins output.
1	Advance to new page; begins output at the top of a new page.
+	Overwrites; begins output at the beginning of the current line and returns to the left margin.
ASCII NUL	Overwrites with no advance; begins output at beginning of the current line and does not return to left margin.
$	*Prompting: begins output at the beginning of the next line and suppresses carriage return at end of line.*

Using FORM='PRINT' specifies formatted and implies vertical format control for that unit. You can use the UNIX utility fpr to interpret the vertical format controls before printing the file.

Chapter 9
Subprograms

Subprograms are program units that can be invoked from other program units. Subprograms usually perform often-used sequences of operations for the invoking program unit. Arguments (dummy and actual) of the subprogram are used to transfer information between the subprogram and another program unit. The dummy argument appears in the argument list of a subprogram and the actual argument appears in the argument list of a subprogram reference.

There are two classes of subprograms—BLOCK DATA subprograms and procedures. Procedure subprograms include both function subprograms and subroutine subprograms. The procedures can be either user-defined or supplied as part of the CONVEX FORTRAN system.

9.1 Dummy and Actual Arguments

Dummy arguments, which are classified as variables, arrays, or dummy procedures, are used in statement functions, function subprograms, and subroutine subprograms to indicate the number and types of actual arguments to be transferred. The dummy arguments indicate whether each actual argument is a single value, array of values, procedure, or statement label. You cannot use a dummy argument name in a DATA, EQUIVALENCE, INTRINSIC, SAVE, or COMMON statement except as a common block name.

Actual arguments, which may be constants, symbolic names of constants, function references, expressions, arrays and array elements, character substrings, alternate return specifiers, or subprogram names, specify the entities that are to be associated with the dummy arguments. The type of each actual argument must agree with the type of its associated dummy argument, except when the actual argument is a subroutine name or an alternate return specifier. Actual arguments must also agree in order and number with the dummy arguments.

A function or subroutine reference establishes an association between the corresponding dummy and actual arguments. The dummy argument holds the value of the actual argument during execution. For example, using the following statements:

```
SUBROUTINE SAMPLE (R,L)
CALL SAMPLE (B,80)
```

specifies R and L as the dummy arguments. The actual arguments (B,80) replace the dummy arguments when the subroutine executes. Thus, B replaces R and 80 replaces L. Any value assigned to R is also assigned to B.

The number of elements of a dummy argument used as an array cannot exceed the number of elements in the actual argument. Also, a type CHARACTER dummy argument length must not be larger than the length of the associated actual argument.

9.1.1 Variables as Dummy Arguments

To associate a dummy argument variable with an actual argument that is a variable, array element, substring, or expression (including a constant), use the variable, array element, substring, or expression as an actual argument and include a dummy argument of the same data type in the subprogram argument list.

You may define the associated dummy argument within the subprogram if the actual argument is a variable name, array element name, or substring name. If the associated actual argument is a constant or constant name, function reference, or an expression, however, it must not be defined within the subprogram. If you pass a constant to a subroutine as an actual parameter and that subroutine attempts to modify the corresponding dummy argument, either by a READ or ASSIGNMENT statement, a segmentation violation occurs because the constants are stored in read-only storage.

9.1.2 Arrays as Dummy Arguments

If a dummy argument is declared as an array, it can only be associated with an actual argument that is an array or array element of the same type. To pass an array to a subprogram, use the array name as the actual argument. The subprogram must dimension the array to use it. That is, the dummy array must be specified in an array declarator in the subprogram. The declarator has the same format as that for an actual array but with the following differences:

- You cannot use the declarator in a COMMON statement. It is, however, permitted in a DIMENSION or type statement.
- Integer constant expressions and expressions containing integer constants and variables can be used as upper and lower bounds of array dimensions. These dimensions are considered adjustable, as one or both of the dimension-bound expressions is a variable. The array is called an adjustable array.
- You can use an asterisk to specify the upper bound of the last dimension. In this case, the array is known as an assumed-size array.

Common block elements should not be passed as dummy arguments if the called routine or any routine that it, in turn, calls accesses that element from the common block.

9.1.2.1 Adjustable Arrays

Adjustable arrays are used to process arrays of different sizes in a single subprogram. The adjustable array dimensions are determined in the reference to the subprogram.

Each dummy argument in the array declarator must be associated with an actual argument when the subprogram is entered. Any variable used in an adjustable dimension or each COMMON variable appearing in a dimension bound expression must have a defined value when the subprogram is entered. The expressions specifying the adjustable dimensions are evaluated when the subprogram is entered. Argument association is not retained through different references to the subprogram. The bound values are determined each time a subprogram is entered.

In the statement:

```
DIMENSION E(I,I), G(5,2*I)
```

E and G are adjustable arrays.

The size of the adjustable array must be less than or equal to the size of the array of the corresponding actual argument.

9.1.2.2 Assumed-Size Arrays

An asterisk is used to specify the upper bound of the last array dimension in an assumed-size array declarator. For example:

```
DIMENSION SAM (*)
```

sets the upper bound to assumed-size for a one-dimensional array. If the array has more than one dimension, only the last dimension can be assumed size. For example:

```
DIMENSION SAM (1:N,1:*)
```

sets the upper bound for a two-dimensional array.

The assumed-size dummy array name cannot appear:

- In an I/O list of a data transfer statement.
- As an internal unit identifier in an I/O statement.
- As a runtime format identifier in an I/O statement.

The size of the dummy array is the size of the actual argument array when the actual argument corresponding to the dummy array is a noncharacter array name. When the actual argument corresponding to the dummy argument is a noncharacter array element name, however, the size of the array is the array size plus one minus the subscript value.

The size of the dummy array is INT(n + 1 -s)/l, when the actual argument is a character array name, character array element name, or character array element substring, and begins at character storage unit *s* of an array with *n* character storage units with *l* as the length of an element of the dummy array. If an assumed-size dummy array has *n* dimensions, the product of the sizes of the first n-1 dimensions must be less than or equal to the size of the array.

9.1.3 Character Arguments

You can use character values in one or more of the dummy arguments in a subprogram if the actual argument in the calling program unit is type CHARACTER. Thus, a dummy argument that is a variable name of type CHARACTER can be associated only with an actual argument that is either a character variable, character array element, character substring, or character expression. A dummy argument that is an array name of type CHARACTER can be associated only with an actual argument that is a character array, character array element, or character array element substring.

If the actual argument is a Hollerith constant, i.e., 3HSAM, the dummy argument must be of numeric data type. The corresponding dummy argument can have either a numeric or character data type when the actual argument is a character constant ("SAM").

9.1.3.1 Character Argument Lengths

The length of the dummy argument must not exceed the length of the actual argument. The subprogram cannot access more characters than are declared for the argument in the calling unit. That is, when the dummy argument is of type CHARACTER, the associated actual argument must be less than or equal to the length of the actual argument. If the length of the dummy argument of type CHARACTER is less than the length of the associated actual argument, only the leftmost characters of the actual argument become associated with the dummy argument.

If you use an assumed-length character argument, it must be a dummy argument. When control transfers to a subprogram, the assumed-length character dummy argument must be associated with a character actual argument. It assumes the length of the corresponding actual argument. Thus, if you use specify the dummy argument length as an assumed-length character argument, i.e, *(*), the length used is the length of the associated actual argument.

If the dummy argument is an array, you can specify a length that differs from that of the calling unit. In this case, however, the subprogram cannot access a character beyond the last character reserved by the calling unit for the array. The restriction on length is for the entire array and not each array element when a dummy argument of type CHARACTER is an array name.

You can also use a character array dummy argument with an assumed-length. In this case, the length of each element in the dummy argument equals the length of the elements in the actual argument. The assumed length and the array declarator determine the size of the assumed-length character array.

The following example illustrates length specified for the arguments:

```
PROGRAM SAM
CHARACTER A1*2, A2*6, A3*8
   ...
END

SUBROUTINE MYEX(A)
CHARACTER A*6
   ...
END
```

Assume the following CALL statements occur in the main program SAM:

```
CALL MYEX (A1)
CALL MYEX (A2)
CALL MYEX (A3)
```

The first statement is invalid because the length of the dummy argument exceeds that of the associated argument; the remaining two statements are valid with the six leftmost characters of A3 being associated with A. If, however, the subprogram MYEX had been defined as:

```
SUBPROGRAM MYEXAM (A)
CHARACTER A*(*)
   ...
END
```

all three of the CALL statements would have been valid. The length of the dummy argument A is determined by the length of the corresponding actual argument.

9.1.4 Procedures as Dummy Arguments

A dummy argument is considered a dummy procedure if the dummy procedure name appears in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement and if:

- it is referenced as a function,
- it appears in a type statement and EXTERNAL statement, or
- it is referenced as a subroutine.

When you use a dummy argument that is a dummy procedure, associate it only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

When you use a dummy argument as if it were an external function, or in a type statement and EXTERNAL statement, the associated actual argument must be an intrinsic function, external function, or dummy procedure.

If you use the dummy argument as a procedure name in a function reference and associate it with an intrinsic function, the arguments must agree in order, number, and type with those specified for the intrinsic function.

When you use the dummy argument as if it were a subroutine, the actual argument must be the name of a subroutine or dummy procedure. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, it is not possible to determine whether the symbolic name becomes associated with a function or subroutine by examination of the subprogram alone.

9.1.5 Alternate Return Arguments

You can use an asterisk as a dummy argument only in the dummy argument list of a SUBROUTINE statement or an ENTRY statement in a subroutine subprogram. When you use the asterisk as a dummy argument, the corresponding actual argument must be an alternate return specifier in the CALL statement. For example:

```
SUBROUTINE EXAM(D3,*,E2,*)
```

The alternate return allows you to return control to any executable labeled statement in the calling program as long as you have included alternate return arguments in the corresponding positions. These actual arguments have the form:

*label or *&label*

9.2 Functions

A function can be an intrinsic function, a statement function, or an external function (function subprogram). These functions all supply a value to the expression. The function is referenced from within another part of the program. When executed, a function has a value, and consequently, a type. The general form of a function reference is:

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

where *func* is the symbolic name of the function or dummy procedure being referenced, and *a* is a list of actual arguments separated by commas. If you do not include arguments, you must still include the enclosing parentheses.

9.2.1 Intrinsic Functions

CONVEX FORTRAN supplies intrinsic functions as a built-in language feature. You can invoke these pre-existing functions by using the function name in any part of a user program; no definition is required.

There are two classes of intrinsic names—generic names and specific names. If you reference a generic intrinsic name, the compiler decides which special intrinsic to invoke based on the type of the actual arguments. When you reference specific names, the arguments to the intrinsic must be of a specific type. For example, the generic intrinsic function, LOG (natural logarithm), can accept arguments of the type REAL, DOUBLE PRECISION and COMPLEX; whereas the specific function, DLOG, can only accept a DOUBLE-PRECISION argument.

Using a generic name generally simplifies function referencing because you can use the function name with more than one type of argument. You must use the appropriate specific name whenever the intrinsic function name is to be used as an actual argument in a subprogram. For either generic or specific functions that require multiple arguments, all arguments must be of the same data type. The compiler does not convert incorrectly typed arguments.

Use of the IMPLICIT statement does not alter the type of intrinsic functions.

An intrinsic reference has the form:

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

where *inf* is an intrinsic name and *a* is the argument on which the function operates.

9.2.2 Built-in Functions

The built-in functions allow a FORTRAN program to pass arguments to a program that is not written in FORTRAN.

9.2.2.1 %REF and %VAL

Two built-in functions—%REF and %VAL—may be used in the argument list of a CALL statement or function reference to change the form of the argument. Such a change is necessary when you need to call subprograms written in languages other than FORTRAN, because the actual argument has to be passed in a form different from that used by FORTRAN. These two functions specify how the argument should be passed to the subprogram.

Although you can use these functions in the actual argument list of a CALL statement or function reference, you cannot use them in any other context. You need not, however, use these built-in functions when invoking a FORTRAN library procedure or a user-supplied subprogram written in FORTRAN. The two built-in argument list functions are (where a is an actual argument):

- *%REF(a). This function passes the argument by reference.*
- *%VAL(a). This function passes the argument as a 32-bit immediate value; an argument shorter than 32 bits is sign-extended to a 32-bit value.*

Table 10-1 shows the FORTRAN argument-passing defaults and the allowed uses of %REF and %VAL.

Table 9-1: Built-in Functions and Defaults for Argument Lists

Data	Functions Allowed			
	Type	*Default*	*%REF*	*%VAL*
Expressions:				
*LOGICAL (*1,2,4)*	*REF*	*Yes*	*Yes*[1]	*Yes*
*LOGICAL*8*	*Ref*	*Yes*	*No*	*No*
*INTEGER (*1,2,4)*	*REF*	*Yes*	*Yes*[1]	*Yes*
*INTEGER*8*	*REF*	*Yes*	*No*	*No*
*REAL*4*	*REF*	*Yes*	*Yes*	*Yes*
*REAL*8*	*REF*	*Yes*	*No*	*No*
COMPLEX	*REF*	*Yes*	*No*	*No*
CHARACTER	*REF*	*Yes*	*No*	*No*
Hollerith	*REF*	*No*	*No*	*No*
Array Name:				
Numeric	*REF*	*Yes*	*No*	*No*
Character	*REF*	*Yes*	*No*	*No*
Procedure Name:				
Numeric	*REF*	*Yes*	*No*	*No*
Character	*REF*	*Yes*	*No*	*No*

[1] *If a logical or integer value occupies less than 32 bits of storage, it is converted to a 32-bit value by sign extension.*

9.2.2.2 %LOC

The %LOC built-in function computes the internal address of a storage element, as in the following example:

%LOC(v)

where v is a variable name, array element name, array name, character substring name, or external procedure name.

*The %LOC built-in function produces an INTEGER*4 value that represents the location of its argument. Addresses from user programs are BYTE addresses and are always negative integer values.*

9.2.3 Statement Functions

The statement function is a nonexecutable single-statement procedure defined by the user. You can reference a statement function only from the program unit in which it is defined. The form is similar to an arithmetic, logical, or character assignment statement. ***Statement function definitions must precede the use of the statement function.*** The statement function returns a single value to the program. The form of a statement function is:

func ([*d* [,*d*]...]) = *exp*

where:

func is the name of the statement function. The type is implied by the implicit naming convention or by a prior type statement. Do not use the name to identify any other entity in the current program unit except a common block.

d represents a variable name called a statement function dummy argument. The dummy argument holds the value of the actual argument during execution. The dummy argument list specifies the order, number, and type of actual argument whose values are used in the function reference. The actual arguments must agree in order, number, and type with the corresponding dummy arguments. (The compiler associates the actual arguments with the dummy arguments of the companion statement definition.)

Each name must be unique in the function definition and must be of the data type of the actual value that replaces it during the function reference. You can use the dummy argument name to identify a variable of the same type, as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement, or as a common block name.

exp is an expression. Each primary of *exp* must be one of the following:

- A constant or symbolic name of a constant
- A variable reference
- An array element reference
- An intrinsic function reference
- A statement function that has been previously defined in the current program unit
- An external function reference
- A dummy procedure reference

A statement function is referenced using its function reference as a primary in an expression. The following example illustrates a statement function and the statement function reference.

```
IAVG(IGR1,IGR2,IGR3) = (IGR1 + IGR2 + IGR3)/3
  ...
ISC = IAVG(IOR,IWR,IAP)
```

When the statement function reference executes, all actual arguments that are expressions are evaluated and actual arguments are associated with the corresponding dummy arguments. (The compiler substitutes the values in the actual arguments for the dummy arguments.) Then the processor evaluates the expression (right side of the statement function statement). Conversion occurs, if necessary, of the resulting value to the type of the statement function according to the usual arithmetic assignment rules; or a change occurs, if necessary, in the length of a character expression value according to the usual character assignment rules. The resulting value is available to the expression that contains the function reference.

In a statement function reference, you can use as an actual argument any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

9.3 Function Subprograms

A function subprogram is a separate program unit that consists of a FUNCTION statement followed by a series of statements that define the computing procedure. The calling program unit references it; the statements execute, and through a RETURN or END statment, a single value returns to the function reference in the calling unit. This value is assigned to the function name.

The FUNCTION statement specifies the name of the function, the dummy arguments used by the function, and can indicate the type of the function value. In logical and numeric functions, the FUNCTION statement, ***including the CONVEX extension *m*** and optional parentheses, is represented by:

[*typ*] FUNCTION *nam* ***[*m]*** [([*d* [,*d*]...])]

where:

typ is one of the logical or numeric data-type specifiers.

nam is the symbolic name of the function subprogram. If you do not specify *typ* nor declare the *nam* in a later type statement, the name implies the data type of the function.

m ***is an unsigned, nonzero integer constant specifying the length of the data type. It must be one of the valid length specifiers for the data type given by typ.***

d is a dummy argument name that can include variable names, array names, or dummy procedure names. All dummy argument names must match the actual arguments in all references in number, order, and type. The dummy name is local to the program unit and must not appear in a DATA, EQUIVALENCE, INTRINSIC, SAVE, or COMMON statement, except as a common block name.

In character functions, the CHARACTER FUNCTION statement, ***with the CONVEX extension of *n*** and optional parentheses, is represented thus:

CHARACTER [**n*] FUNCTION *nam* ***[*n]*** [([*d* [,*d*]...])]

where n is either an unsigned, nonzero integer constant, or a parenthetical asterisk (*) indicating an assumed-length function name. If you specify CHARACTER*(*), the function always assumes the length declared for it in the program unit that invokes it. (An assumed-length character function can have different lengths when invoked by different program units.) If *n* is an integer constant, the value of *n* must agree with the length of the function specified in the program unit that invokes the function. If you do not specify *n*, a length of 1 is assumed. If the length has already been specified after the keyword CHARACTER, you cannot use the optional-length specification following *nam*. Both *nam* and *d* retain the same definition.

You must begin the function subprogram with the FUNCTION statement. You can include any statements except a BLOCK DATA, SUBROUTINE, PROGRAM, or another FUNCTION statement within the function subprogram. End it with an END statement. Between a FUNCTION and END statement, you can use the specified function name as a variable in an executable statement or in a type statement if you omit *typ*. Include ENTRY statements to provide multiple entry points to the subprogram.

A function specified in a subprogram can reference other subprograms but cannot reference itself, directly or indirectly. The function must assign a value to its symbolic name at least once.

9.4 Subroutine Subprograms

A subroutine subprogram (subroutine) is a program unit that performs a specific, user-defined task or subtask for some other program unit of the program. Subroutines are similar to function subprograms, as actual and dummy arguments are handled the same for both. The RETURN statement returns control to the calling program. They differ in that the subroutine names have no type and no value is associated with the subroutine name. Also, you use a CALL statement, not a function reference, to invoke a subroutine. Within the subroutine, you can specify different points of return to the calling subprogram. The CALL statement has the form:

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

where:

sub is the name of the subroutine.

a is the actual argument that can be a constant, variable, expression, array, array element, character substring, alternate return specifier, intrinsic function name, external procedure name, or dummy procedure name. You can use an * or & followed by the label of an executable statement to indicate an alternate return. Do not use a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. If the actual argument is a Hollerith constant, however, the dummy argument must be of numeric data type.

Using the CALL statement invokes the subroutine. Control passes to the first executable statement using any *a* arguments for the subroutine dummy arguments. After the subroutine returns, control returns to the statement in the calling unit that follows the CALL statement unless you specify an alternate RETURN in the subroutine.

You must begin a subroutine with a SUBROUTINE statement and terminate it with an END statement. It specifies the name of the subroutine and the arguments used by the subroutine. The SUBROUTINE statement has the form:

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

where:

name in the symbolic name of the subroutine. As the subroutine has no data type, you need not apply the naming rules. Since the name is global, do not use it for any other purpose in the program.

d represents a dummy argument list consisting of a variable name, array name, dummy procedure name, or an asterisk (*), if the subroutine uses alternate returns, separated by commas. The argument list can be empty. In this case, use of parentheses is optional; for example, either SUBROUTINE EXAM or SUBROUTINE EXAM() is acceptable.

If you indicate the dummy argument as *, be sure that the corresponding actual argument in the calling unit is also an * or & followed by the label of an executable statement within the calling unit. You can specify an alternate return in the RETURN statement by giving the position of this asterisk among other asterisks in the dummy argument.

You must specify an actual argument for each dummy argument in the SUBROUTINE statement of the called subroutine. If you use a variable, array element, or array as the actual argument, the data type must match that of the dummy argument. If the argument is the name of a subprogram, you must declare this name in an EXTERNAL statement in this program unit.

The ENTRY statement may be used to specify multiple entry points for subroutines.

9.5 ENTRY Statement

You can use the nonexecutable ENTRY statement to specify alternative entry points into a function or subroutine subprogram. You can reference an ENTRY from any program unit except the subprogram that contains it. Use a function reference for ENTRY in a function; use CALL for ENTRY in a subroutine. You can place an ENTRY anywhere between the initial FUNCTION or SUBROUTINE statement and the END statement, but you must not place it within a block IF or the range of a DO loop. The form of the ENTRY statement is:

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

where:

nam is the symbolic name of the entry point representing either a subroutine name in a subroutine or an external function name in a function subprogram. When entry is in a function, ***nam*** has a data type that you can imply or specify. If you use a type statement, it can appear before or after the ENTRY statement. For entry in a subroutine, however, there is no data type restriction.

d is a variable name, array name, or dummy procedure. You can use an asterisk (*) as an alternate return only if the entry is in a subroutine. You can omit the parentheses for an empty argument list in a subroutine entry, but the parentheses must always be included in a function entry and in the entry reference.

You can use dummy arguments in ENTRY statements that differ in order, number, type, and name from the dummy arguments you use in the FUNCTION, SUBROUTINE, or other ENTRY statements in the same subprogram. Each reference to a function or subroutine, however, must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

If a dummy argument is not currently associated with an actual argument, it is undefined. The actual-dummy argument association is not retained between references.

Defining any entry name or the name of the function subprogram defines all the associated names of the same data type as all entry names within a function subprogram are associated with the name of the function subprogram. You can use function and entry names of different data types.

You must define the variable whose name is used to reference the function before a RETURN or END statement appears in the subprogram. You don't need to use associated variables of the same type unless the function is type CHARACTER; but an associated variable of different type must not become defined within the subprogram.

9.6 RETURN Statement

The RETURN statement is used to return from a subroutine subprogram to one of several alternative points in the calling program unit. You can use none, one, or more than one RETURN statement in a subroutine. Use the alternate return only with subroutine subprograms, not function subprograms. You can, however, use the RETURN statement without an alternate specifier, in either a function or subroutine subprogram. The statement has the form:

RETURN [*e*]

where *e* is an optional integer expression that specifies an alternate statement in the calling program is to receive control. ***The system converts the value type to integer if necessary.*** The *e* represents the number, such as RETURN 2, of the corresponding asterisk, among other asterisks, in the dummy argument list of the subroutine. The alternate return specifier has the form of an asterisk or ampersand followed by the label of an executable statement, e.g., *30 or ***&30***. For example:

```
        ...
      CALL EXAM(D,*30,E,*40)
        ...
30          .           !RETURN 1 goes here.
        ...
40                      !RETURN 2 goes here.
        ...
      END

SUBROUTINE EXAM(D3,*,E2,*)
        ...
RETURN           !Returns after the CALL statement.
        ...
RETURN 1         !Returns to 30.
        ...
RETURN 2         !Returns to 40.
        ...
END
```

RETURN 1 indicates control transfers to the statement at line 30; RETURN 2 indicates the alternate return transfers control to the statement at line 40. RETURN indicates control transfers to the statement immediately after the CALL statement.

If you do not specify a RETURN, the END statement has the same effect as the RETURN.

Thus, using the alternate RETURN statement allows you to return control to any labeled statement in the calling program whose label you specify as an alternate return specifier to the subprogram. If *e* is less than 1 or greater than the total number of asterisks appearing in the dummy argument list, control returns as for a normal RETURN (without specifier).

When a RETURN or END statement executes, the subprogram terminates the association between the dummy arguments and the current actual arguments. All entities within the subprogram become undefined except for entities saved by SAVE statements, entities declared in blank common, initially defined entities that have neither been redefined or become defined, and entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referencing the subprogram (directly or indirectly).

When used from a function, RETURN transfers control to the function reference in the calling unit and returns the function value. In a subroutine, RETURN transfers control to the statement following the CALL statement in the calling program unit.

Chapter 10
Block Data Subprogram

You can use a block data subprogram to provide initial values for variables and array elements in named common blocks. A block data subprogram is nonexecutable; it must not contain any executable statements. The block data subprogram has the nonexecutable BLOCK DATA statement as its first statement. You can use only one BLOCK DATA statement in a subprogram, but you can use more than one block data subprogram in the program units that constitute the executable program. The statement has the form:

BLOCK DATA [*name*]

where *name* is an optional symbolic name for the block data subprogram in which the BLOCK DATA statement appears. Do not assign the BLOCK DATA the same name as for an external procedure, main program, common block, or other block data subprogram, nor any local name in the block data subprogram.

You can use only these specification statements between the BLOCK DATA and END statements: COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and any of the type-declaration statements. Your block data subprogram must contain at least one COMMON statement and one DATA statement.

All entities having storage units in the common block must be specified, but you are not required to initialize all of the values. Be sure to provide specifications to establish the entire block.

Example:

```
BLOCK DATA Sample
COMMON/EX/A,B,C
COMMON/CAT/LIST(100)
DATA A/3.5/,LIST/(100*5)
```

A

Generic and Intrinsic Functions

This appendix lists the generic and intrinsic functions for CONVEX FORTRAN. The numbers in the first column refer to the notes following the table. Operand types required for intrinsic functions are not changed when the *-i* or *-r* compiler option is used. The correct specific routines must be called when default data sizes are changed.

Table A-1: Generic and Intrinsic Functions

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
SQRT 1	SQRT DSQRT CSQRT *CDSQRT*	Square Root a**1/2	1	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*
LOG 2	ALOG DLOG CLOG *CDLOG*	Natural Logarithm log(e)a	1	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*
LOG10 2	ALOG10 DLOG10	Common Logarithm log(10)a	1	REAL*4 REAL*8	REAL*4 REAL*8
EXP	EXP DEXP CEXP *CDEXP*	Exponential e**a	1	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*
SIN 3	SIN DSIN CSIN *CDSIN*	Sine Sin a	1	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*
SIND 3	*SIND* *DSIND*	*Sine(degree)* *Sin a*	*1*	*REAL*4* *REAL*8*	*REAL*4* *REAL*8*
COS 3	COS DCOS CCOS *CDCOS*	Cosine Cos a	1	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*
COSD 3	*COSD* *DCOSD*	*Cosine(degree)* *Cos a*	*1*	*REAL*4* *REAL*8*	*REAL*4* *REAL*8*
TAN 3	TAN DTAN	Tangent Tan a	1	REAL*4 REAL*8	REAL*4 REAL*8

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
TAND 3	TAND DTAND	Tangent(degree) Tan a	1	REAL*4 REAL*8	REAL*4 REAL*8
ASIN 4,5	ASIN DASIN	Arc Sine Arc Sin a	1	REAL*4 REAL*8	REAL*4 REAL*8
ASIND 2,4,5	ASIND DASIND	Arc Sine(degree) Arc Sine a	1	REAL*4 REAL*8	REAL*4 REAL*8
ACOS 4,5	ACOS DACOS	Arc Cosine Arc Cos a	1	REAL*4 REAL*8	REAL*4 REAL*8
ACOSD 2,4,5	ACOSD DACOSD	Arc Cosine(degree) Arc Cos a	1	REAL*4 REAL*8	REAL*4 REAL*8
ATAN 5	ATAN DATAN	Arc Tangent Arc Tan a	1	REAL*4 REAL*8	REAL*4 REAL*8
ATAND 2,5	ATAND DATAND	Arc Tangent(degree) Arc Tan a	1	REAL*4 REAL*8	REAL*4 REAL*8
ATAN2 5,6	ATAN2 DATAN2	Arc Tangent Arc Tan a(1)/a(2)	2	REAL*4 REAL*8	REAL*4 REAL*8
ATAN2D 2,5,7	ATAN2D DATAN2D	Arc Tangent(degree) Arc Tan a(1)a(2)	2	REAL*4 REAL*8	REAL*4 REAL*8
SINH	SINH DSINH	Hyperbolic Sine Sinh a	1	REAL*4 REAL*8	REAL*4 REAL*8
COSH	COSH DCOSH	Hyperbolic Cosine Cosh a	1	REAL*4 REAL*8	REAL*4 REAL*8
TANH	TANH DTANH	Hyperbolic Tangent Tanh a	1	REAL*4 REAL*8	REAL*4 REAL*8
ABS 8	IIABS JIABS KIABS ABS DABS CABS CDABS	Absolute Value |a|	1	INTEGER*2 INTEGER*4 INTEGER*8 REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	INTEGER*2 INTEGER*4 INTEGER*8 REAL*4 REAL*8 REAL*4 REAL*8
IABS 8	IIABS JIABS KIABS	Absolute Value |a|	1	INTEGER*2 INTEGER*4 INTEGER*8	INTEGER*2 INTEGER*4 INTEGER*8

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
INT 9,14 15	*IINT* *JINT* *KINT* *IIDINT* *JIDINT* *KIDINT*	*Truncation* *[a]*	*1*	*REAL*4* *REAL*4* *REAL*4* *REAL*8* *REAL*8* *REAL*8* *COMPLEX*8* *COMPLEX*8* *COMPLEX*8* *COMPLEX*16* *COMPLEX*16* *COMPLEX*16*	*INTEGER*2* *INTEGER*4* *INTEGER*8* *INTEGER*2* *INTEGER*4* *INTEGER*8* *INTEGER*2* *INTEGER*4* *INTEGER*8* *INTEGER*2* *INTEGER*4* *INTEGER*8*
10,14	*INT1* *INT2* *INT4* *INT8*	*Conversion to Integer*	*1* *1* *1* *1*	*Any numeric* *Any numeric* *Any numeric* *Any numeric*	*INTEGER*1* *INTEGER*2* *INTEGER*4* *INTEGER*8*
IDINT 9,14 15	*IIDINT* *JIDINT* *AKIDINT*	Truncation [a]	1	*REAL*8* REAL*8 *REAL*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
AINT	AINT DINT	Truncation [a]	1	REAL*4 REAL*8	REAL*4 REAL*8
NINT 9,14 15	*ININT* *JNINT* *KNINT* *IIDNNT* *JIDNNT* *KIDNNT*	Nearest Integer [a+.5*sign(a)]	1	*REAL*4* REAL*4 *REAL*4* *REAL*8* *REAL*8* *REAL*8*	*INTEGER*2* INTEGER*4 *INTEGER*8* *INTEGER*2* *INTEGER*4* *INTEGER*8*
IDNINT 9,14 15	*IIDNNT* *JIDNNT* *KIDNNT*	Nearest Integer [a+.5*sign(a)]	1	*REAL*8* REAL*8 *REAL*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
ANINT 9,15	ANINT DNINT	Nearest Integer [a+.5*sign(a)]	1	REAL*4 REAL*8	REAL*4 REAL*8

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
ZEXT 14,15	*IZEXT* *JZEXT* *KZEXT*	*Zero-Extend Functions*	*1*	*LOGICAL*1* *LOGICAL*2* *INTEGER*2* *LOGICAL*1* *LOGICAL*2* *LOGICAL*4* *INTEGER*2* *INTEGER*4* *LOGICAL*1* *LOGICAL*2* *LOGICAL*4* *LOGICAL*8* *INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
REAL 10	*FLOATI* *FLOATJ* *FLOATK* SNGL	*Conversion to REAL*4*	*1*	*INTEGER*2* *INTEGER*4* *INTEGER*8* *REAL*4* REAL*8 COMPLEX*8 *COMPLEX*16*	*REAL*4* *REAL*4* *REAL*4* *REAL*4* REAL*4 REAL*4 *REAL*4*
DBLE 10	 DBLE	Conversion to REAL*8	1	INTEGER*2 INTEGER*4 INTEGER*8 REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	REAL*8 REAL*8 REAL*8 REAL*8 REAL*8 REAL*8 *REAL*8*
IFIX 10,14 15	*IIFIX* *JIFIX* *KIFIX*	FIX (REAL*4-to-integer conversion)	1	*REAL*4* REAL*4 *REAL*4*	*INTEGER*2* INTEGER*4 *INTEGER*8*
FLOAT 10	*FLOATI* *FLOATJ* *FLOATK*	Float (Integer-to-REAL*4 conversion)	1	*INTEGER*2* INTEGER*4 *INTEGER*8*	*REAL*4* REAL*4 *REAL*4*

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
DFLOAT 10	*DFLOTI* *DFLOTJ* *DFLOTK*	REAL*8 Float (Integer-to-REAL*8 conversion)	1	*INTEGER*2* INTEGER*4 *INTEGER*8*	*REAL*8* REAL*8 *REAL*8*
CMPLX 11,15		Conversion to COMPLEX*8 or COMPLEX*8 from Two Arguments	1,2 1,2 1,2 1,2 1,2 1 1	*INTEGER*2* INTEGER*4 INTEGER*8 REAL*4 REAL*8 COMPLEX*8 *COMPLEX*16*	*COMPLEX*8* COMPLEX*8 COMPLEX*8 COMPLEX*8 COMPLEX*8 COMPLEX*8 *COMPLEX*8*
DCMPLX 11,15		*Conversion to COMPLEX*16* *or* *COMPLEX*16 from Two Arguments*	*1,2* *1,2* *1,2* *1,2* *1,2* *1* *1*	*INTEGER*2* *INTEGER*4* *INTEGER*8* *REAL*4* *REAL*8* *COMPLEX*8* *COMPLEX*16*	*COMPLEX*16* *COMPLEX*16* *COMPLEX*16* *COMPLEX*16* *COMPLEX*16* *COMPLEX*16* *COMPLEX*16*
	REAL *DREAL*	Real Part of Complex	1	COMPLEX*8 *COMPLEX*16*	REAL*4 *REAL*8*
	AIMAG *DIMAG*	Imaginary Part of Complex	1	COMPLEX*8 *COMPLEX*16*	REAL*4 *REAL*8*
CONJG	CONJG *DCONJG*	Complex Conjugate *(if A =(X,Y)* *CONJG(A)=(X,-Y))*	1	COMPLEX*8 *COMPLEX*16*	COMPLEX*8 *COMPLEX*16*
	DPROD	REAL*8 Product of REAL*4 a(1)*a(2)	2	REAL*4	REAL*8
MAX 12,15	*IMAX0* *JMAX0* *KMAX0* AMAX1 DMAX1	Maximum max(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8
MAX0 12,15	*IMAX0* *JMAX0* *KMAX0*	Maximum max(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
MAX1 12,14 15	*IMAX1* *JMAX1* *KMAX1*	Maximum max(a(1),a(2),...a(n))	n	*REAL*4* REAL*4 *REAL*4*	*INTEGER*2* INTEGER*4 *INTEGER*8*
AMAX0 12,15	*AIMAX0* *AJMAX0*	Maximum max(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4	*REAL*4* REAL*4

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
MIN 13,15	*IMINO* *JMINO* *KMINO* AMIN1 DMIN1	Minimum min(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8
MIN0 13,15	*IMINO* *JMINO* *KMINO*	Minimum min(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
MIN1 13,14 15	*IMIN1* *JMIN1* *KMINI*	Minimum min(a(1),a(2),...a(n))	n	*REAL*4* REAL*4 *REAL*4*	*INTEGER*2* INTEGER*4 *INTEGER*8*
AMIN0	*AIMINO* *AJMINO*	Minimum min(a(1),a(2),...a(n))	n	*INTEGER*2* INTEGER*4	*REAL*4* REAL*4
DIM	*IIDIM* *JIDIM* *KIDIM* DIM DDIM	Positive Difference a(1)-(min(a(1),a(2)) (Returns the first argument minus the minimum of the two arguments)	2	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8
IDIM	*IIDIM* *JIDIM* *KIDIM*	Positive Difference a(1)-(min(a(1),a(2)) (Returns the first argument minus the minimum of the two arguments)	2	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
MOD	*IMOD* *JMOD* *KMOD* AMOD DMOD	Remainder a(1)-a(2)*[a(1)/a(2)] (Returns the remainder when the first argument is divided by the second)	2	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8
SIGN	*IISIGN* *JISIGN* *KISIGN* SIGN DSIGN	Transfer of Sign |a(1)|Sign a(2)	2	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8	*INTEGER*2* INTEGER*4 *INTEGER*8* REAL*4 REAL*8
ISIGN	*IISIGN* *JISIGN* *KISIGN*	Transfer of Sign |a(1)|Sign a(a)	2	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
IAND	*IIAND* *JIAND* *KIAND*	*Bitwise AND (performs a logical AND on corresponding bits)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
IOR	*IIOR* *JIOR* *KIOR*	*Bitwise OR (performs an inclusive OR on corresponding bits)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
IEOR	*IIEOR* *JIEOR* *KIEOR*	*Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
NOT	*INOT* *JNOT* *KNOT*	*Bitwise Complement (complements each bit)*	*1*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
ISHFT 16*	*IISHFT* *JISHFT* *KISHFT*	*Bitwise Shift (a(1) logically shifted a(2) bits—positive a(2) argument shifts left; negative, right)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
IBITS† *17*	*IIBITS* *JIBITS* *KIBITS*	*Bit Extraction (extracts bits a(2) through a(2)+a(3)-1 from a(1));*	*3*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
IBSET†	*IIBSET* *JIBSET* *KIBSET*	*Bit Set (returns the value of a(1) with bit a(2) of a(1) set to 1)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
BTEST†	*BITEST* *BJTEST* *BKTEST*	*Bit Test (returns. TRUE. if bit a(2) of argument a(1) equals 1)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*LOGICAL*2* *LOGICAL*4* *LOGICAL*8*
IBCLR†	*IIBCLR* *JIBCLR* *KIBCLR*	*Bit Clear (returns the value of a(1) with bits a(2) of a(1) set TO 0)*	*2*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*
ISHFTC†	*IISHFTC* *JISHFTC* *KISHFTC*	*Bitwise Circular Shift (circularly shifts rightmost a(3) bits of argument a(1) by a(2) places)*	*3*	*INTEGER*2* *INTEGER*4* *INTEGER*8*	*INTEGER*2* *INTEGER*4* *INTEGER*8*

†Arguments after the first are converted to the type of the first argument

Table A-1: Generic and Intrinsic Functions (continued)

Generic	Specific	Functions	No. of Arg.	Type of Argument	Type of Result
15	LEN	Length (returns length of the character expression)	1	CHARACTER	INTEGER*4
15	INDEX	Index(C(1)C(2)) (returns the position of the substring c(2) in the character expression c(1))	2	CHARACTER CHARACTER	INTEGER*4
15	CHAR	Character (returns a character that has the ASCII value specified by the argument)	1	*LOGICAL*1* *INTEGER*2* INTEGER*4 *INTEGER*8*	CHARACTER
15	ICHAR	ASCII Value (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	CHARACTER	INTEGER*4
	LLT LLE LGT LGE	Character relationals (ASCII collating sequence)	2	CHARACTER CHARACTER CHARACTER CHARACTER	LOGICAL*4 LOGICAL*4 LOGICAL*4 LOGICAL*4

NOTES:

1. The SQRT or DSQRT argument must be greater than or equal to zero. The result is the principal value where the real part is greater than or equal to zero. If the real part is zero, the result is the principal value with the imaginary part greater than or equal to zero.

2. The ALOG, DLOG, ALOG10, DLOG10, *ATAND, ATAN2D, ASIND, DASIND, ACOSD,* or *DACOSD* argument, must be greater than zero. The CLOG or *CDLOG* argument cannot be (0.,0.).

3. The SIN, DSIN, COS, DCOS, TAN or DTAN argument must be in radians, and is treated modulo 2*pi. ***The SIND, COSD, or TAND argument must be in degrees, and the argument is treated modulo 360. The value of the sine and cosine functions for very large arguments is unmeaningful. This is related to the accuracy of the argument reduction. For single precision, the maximum value is*** $\pi * 2^{23}$. For double precision, the maximum value is $\pi * 2^{52}$. If the argument exceeds the maximum, it is replaced with zero and evaluation continues.

4. The absolute value of the ASIN, DASIN, ACOS, DACOS, *ASIND, DASIND, ACOSD,* or *DACOSD* argument must be less than or equal to 1.

5. The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, or DATAN2 is in radians, and that of *ASIND, DASIND, ACOSD, DACOSD, ATAND, DATAND, ATAN2D,* or *DATAN2D* is in degrees.

6. If the value of the first ATAN2 or DATAN2 argument is positive, the result is positive; if it is zero, the result is zero if the second argument is positive, and pi if it is negative. A negative value for the first argument determines a negative result. A zero value for the second argument results in the absolute value of pi/2. Neither argument can have the value zero. The range of the result for ATAN2 or DATAN2 is -pi < result ≤ pi.

7. ***If the value of the first ATAN2D or DATAN2D argument is positive, the result is positive. If it is zero, the result is zero if the second argument is positive, and 180 degrees if it is negative. A negative value for the first argument means the result is negative. A zero value for the second argument results in the absolute value of 90 degrees. Neither argument can have the value zero. The range of the result for ATAN2 or DTAN2D is: -180 degrees < result ≤ 180 degrees.***

8. The absolute value of a complex number, (X,Y), is the real value: (X**2 + Y**2)**1/2

9. Define [x] as the largest integer whose magnitude is not greater than the magnitude of x, and whose sign matches that of x. For example [5.7] equals 5. and [-5.7] equals -5.

10. Functions used to convert one data type to another have the same effect as the implied conversion in assignment statements. The functions REAL with a real argument, DBLE with a double-precision argument, and INT with an integer argument return the value of the argument without conversion.

11. If CMPLX or *DCMPLX* has only one argument, the argument converts into the real part of a complex value, and zero is assigned to the imaginary part. If there are two arguments (not complex), conversion of the first argument into the real part of the value, and the second argument into the imaginary part, produces a complex value.

12. This function causes the return of the maximum value from the argument list; there must be at least two arguments.

13. This function causes the return of the minimum value from the argument list; there must be at least two arguments.

14. This function converts to the default integer type.

15. ***The INT, IDINT, NINT, IDNINT, IFIX, MAX1, MINI, and ZEXT functions return INTEGER*4 values if the /I4 or -i4 flag is in effect, INTEGER*2 values if the /NOI4 or i2 or /I2 flag is in effect, or INTEGER*8 values if the /I8 or -i8 flag is in effect.***

16. These functions shift binary patterns—positive, left (a(2) is >0) and negative, right (a(2) < 0). Since ISHFT indicates a logical shift, the bits shifted out of one end are lost and zeros are shifted in at the other end. As ISHFTC specifies a circular shift, the bits shifted out at one end are shifted back in at the other end.

17. A bit in a binary pattern has a value of 0 or 1, being numbered right (least significant—0) to left (most significant—63). A bit field is an adjacent group of bits within a binary pattern with a specified starting bit position and length.

B

Compiler Options

B.1 Compiling Programs

To invoke the CONVEX FORTRAN compiler, use the following command line:

fc [*options*] *files* [*loader-options*]

In the command line, ***options*** is one or more of the compiler options described in the following sections. Any options contained in an OPTIONS statement within a program override those specified on the command line.

The parameter *files* represents one or more FORTRAN source files to be compiled, object files to be loaded, or symbolic assembly language files to be assembled.

The parameter *loader-options* is one or more loader options as described in the *CONVEX Loader User's Guide*. If specified, these options are passed to the UNIX loader when compilation is complete.

Language-Compatibility Options

-cfc Causes the compiler to use the Cray FORTRAN language definition instead of the standard CONVEX FORTRAN definition. This option cannot be used with the *-i* or *-r* options.

-F66 Selects FORTRAN-66 language interpretation rules in cases of incompatibility.

-sa Prevents FORTRAN from generating pre-compiled argument packets in the text segment. All arguments are placed on the stack. This option should only be used when an application contains user-supplied C programs called from FORTRAN. Using it with applications coded only in FORTRAN slows down the application.

-sfc Causes the compiler to use the available subset of Sun f77 language features instead of the corresponding CONVEX FORTRAN features. The subset is described in Appendix I of the *FORTRAN Language Reference Manual.*

-vfc Causes the compiler to accept certain language extensions implemented in VAX FORTRAN instead of the corresponding CONVEX FORTRAN features.

Optimization Options

-ep *n*
Specifies the expected number of processors (*n*) on which the program is going to run. If the value of *n* is not an integer from 1 to 4, the behavior of the compiler is indeterminate.

The compiler parallelizes a loop whenever doing so appears to decrease the turnaround time, assuming the given number of processors. Use this option with caution since it may lead to inefficient use of processors.

-il
Instructs the compiler to prepare an intermediate language (*.fil*) file for a subprogram that is to be used for inline substitution. The *-il* option cannot be used with the *-c*, *-cs*, or *-S* options. Optimization levels are ignored.

-is *directory*
Instructs the compiler to perform inline substitution of each subprogram for which there exists a *.fil* file in the specified *directory*. This option must be repeated for each directory containing *.fil* files to be used for inline substitution.

-no
Specifies that the compiler is to perform no optimization. This option is the default if the -O option is not specified.

-O*n*
Performs machine-independent optimizations at the specified level. You can specify the following optimization levels:

Level	Description
-O0	Local scalar optimization
-O1	Local scalar optimization and global scalar optimization
-O2	Local scalar optimization, global scalar optimization, and vectorization
-O3	Local scalar optimization, global scalar optimization, vectorization, and parallelization

If this option is not used, the compiler performs no machine-independent optimization.

-rl
Causes the compiler to perform loop-replication optimizations on loops selected by the compiler on the basis of profitability. The loop-replication options include loop unrolling and dynamic loop selection.

This option may not be used unless the -O2 option is also specified. When the -O2 option is specified, the compiler generates scalar and vector versions of eligible loops and selects the best version at runtime.

-uo
Performs potentially unsafe optimizations, e.g., moves the evaluation of common subexpressions and/or invariant code from within conditionally executed code. Such moved code may be executed unconditionally.

Code-Generation Options

-c
Suppresses the loading phase of the compilation. For example, output from the file *file.f* or *file.s* is written to *file.o*.

-fi Specifies that real constants are to be translated into IEEE format and processed in IEEE mode. If you specify this option, your machine must be equipped with the IEEE support hardware, or an error message occurs and compilation terminates. If you do not specify a floating-point format, your site default is used.

NOTE: The CONVEX hardware and software only support the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

-fn Specifies that real constants are to be translated into native format and processed in native mode. If you do not specify a floating-point format, your site default is used.

-i*n* Controls how the compiler interprets INTEGER and LOGICAL declarations with unspecified lengths. The default interpretation is INTEGER*4 and LOGICAL*4. The option changes the interpretation to INTEGER*2 and LOGICAL*2, or INTEGER*8 and LOGICAL*8. *n* may be 2, 4, or 8.

-r*n* Controls how the compiler interprets REAL declarations having unspecified lengths. The default interpretation is REAL*4. *n* may be 4 or 8.

-re Causes the compiler to generate reentrant code for parallel or recursive invocation of subprograms. This option makes it possible to call subroutines from inside parallel loops.

Each invocation of a subroutine has its own copy of local variables. Arguments are passed on the stack instead of by means of argument packets. Common variables and saved or initialized variables are still shared among invocations.

If you compile a program using the *-re* option, you must initialize all local variables.

-S Generates symbolic assembly code for each program unit in a source file. Assembler output for source *myfile.f* is written to *myfile.s*. The assembly file is not assembled to produce object code.

-tm *target* Specifies the target machine architecture for which compilation is to be performed. The value for *target* can either be c1 or c2 (C1 or C2 may also be used). If you specify a target machine, the instruction set for that machine is used regardless of the machine on which the compiler is running. If you do not specify a target machine, the compiler generates instructions for the class of machine on which it is running.

Debugging and Profiling Options

-a1 Causes noncharacter arrays declared with a last dimension of 1 to be treated as if they were declared assumed-size (last dimension of *). Subscript checking can then be performed if the *-cs* option is also specified. The *-a1* option can be used in the OPTIONS statement.

-cs Compiles code to check that each subscript is within its array bounds. Does not check the bounds for arrays that are dummy arguments for which the last dimension bound is specified as * or 1. The *-cs* option can be used in the OPTIONS statement.

-db Produces additional information for use by the symbolic debugger, *csd,* and passes the *-lg* option to the loader. This option can be used with all levels of optimization. If the -O option is specified, there may be source statements for which no debugging information is generated for *csd.*

-dc Specifies that a line with a D in column 1 is to be compiled and not treated as a comment line. Statements with a D in column 1 can be conditionally compiled, making this feature a useful debugging tool.

-p Produce code that counts the number of times each routine is called. If loading takes place, the standard startup routine is replaced with one that automatically calls *monitor* at the start and arranges to write out a *mon.out* file at normal termination of the object program.

Also, a profiling library is searched instead of the standard FORTRAN library. An execution profile can then be generated by use of *prof* (optional product).

-pb Causes the compiler to produce source-level counting code that produces an execution profile named *bmon.out* at normal termination. Listings of source-level execution counts can then be obtained with the use of *bprof* (optional product).

-pg Causes the compiler to produce counting code in the manner of *-p* but invokes a runtime recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. An execution profile can then be generated by use of *gprof* (optional product).

-sc Provides a syntax check. Stops compilation of each program unit in a source file after the program has been determined to be a valid FORTRAN program. Using this option during program development reduces compilation times.

Message and Listing Options

-na Suppresses all advisory diagnostic messages.

-nv This option is no longer available. Use *-or* instead.

-nw Suppresses all warning diagnostic messages.

-or *table* Specifies the contents of the optimization report to be produced; either the loop table, the array table, or both, can be displayed. The value for *table* can be *all, none, loop,* or *array.* If this option is not specified, only the loop table is displayed. Section 1.6.2 of this manual describes the optimization report.

-xr

Calls the *fxref* cross-reference generator. The following options are related to this option:

Option	Description
-iw *n*	Specify the column width for identifiers. *n* can range from 8 to 32. The default is 16.
-pw *n*	Specify the logical page width used by the output formatter. The default is 132.
-sl	Produce a source listing with line numbers that precedes the cross-reference table.

-xrl

Calls the *fxref* cross-reference generator and puts all objects (such as variables and arrays) into one table, rather than printing a separate table for each class of objects.

Miscellaneous Options

-B*string*

Finds the substitute compiler (*fskel* and *fpp*) in the directory named *string*. The default directory is */usr/convex/oldfc*, which contains the previous version of the compiler for use as a backup.

-o *name*

Assigns *name* as the name of the executable file produced by the loader. The default name is *a.out*. If the loader is not invoked because the *-c* option is specified and if there is only one file to compile or assemble, then *name* becomes the name of the object module.

-tl *n*

Sets the maximum CPU time limit for compilation to *n* minutes. If the time limit is exceeded, compilation terminates with the message *System error in /usr/convex/fskel*.

-vn

Display information concerning the version of the compiler that is being used. Output goes to *stderr*.

-72

Causes the compiler to process only the first 72 characters of each program line. (The compiler normally processes all characters.) Continued Hollerith and character constants are not padded. A line with fewer than 72 characters ending with a Hollerith constant is padded with blanks until the constant is completed, or until 72 characters are processed for that line. A line with fewer than 72 characters ending with the first characters of a character constant is padded with blanks until 72 characters have been processed. A tab counts as one character.

C

Compiler Directives

Some directives provide information to the compiler that it cannot deduce on its own. Other directives instruct the compiler to override certain default conditions that control optimization, vectorization, or parallelization. A directive line has the following format:

C$DIR *directive* [, *directive*]

The line begins with the characters C$DIR followed by one or more of the directives described in this appendix. If two or more directives are specified, they are separated by commas.

A directive must fit on one line; it cannot be continued. A directive may be surrounded by any number of comment lines.

C.1 Information Directives

The information directives provide information to the compiler and may or may not cause the compiler to take any action. The information directives are

- MAX_TRIPS
- NO_RECURRENCE
- NO_SIDE_EFFECTS

C.1.1 MAX_TRIPS Directive

The MAX_TRIPS directive tells the compiler that the loop which follows is never executed more than the specified number of times. The format of this directive is

MAX_TRIPS (*n*)

This directive can be used to prevent strip mining, when it might otherwise be performed, by specifying a value of *n* that is less than the vector register length of 128.

C.1.2 NO_RECURRENCE Directive

The NO_RECURRENCE directive instructs the compiler to disregard an apparent recurrence in a loop. If there is no other impediment to vectorization, the loop is vectorized. The format of this directive is

NO_RECURRENCE

This directive must be placed immediately before a DO statement or a labeled statement that begins a loop. Comment lines can appear between the directive and the beginning of the loop.

The NO_RECURRENCE directive does not affect recurrences caused by a nested DO loop. The directive can, however, be used on each loop in a nest to give the vectorizer maximum opportunity for improving the performance of the nest.

When the NO_RECURRENCE directive is used, the compiler breaks the recurrence by arbitrarily removing one or more dependencies of the cycle. In the following example, if J is positive, there is no recurrence.

```
C$DIR NO_RECURRENCE
      DO 10 I + 1,N
10    A(I) = A(I+J)
```

The compiler always processes a NO_RECURRENCE directive when the apparent recurrence involves an array element; the compiler always ignores a NO_RECURRENCE directive when the apparent recurrence involves a scalar. In the latter case, the compiler knows that a recurrence exists.

NOTE

Incorrect results may occur if you mistake a real recurrence for an apparent one. Always test vector results against scalar results to determine whether a recurrence is real or apparent.

C.1.3 NO_SIDE_EFFECTS Directive

The NO_SIDE_EFFECTS directive instructs the compiler that the specified functions do not modify the value of a parameter or common variable, perform a read or write, or call another routine. The format of this directive is

NO_SIDE_EFFECTS [(*func* [, *func*])]

The parameter *func* specifies one or more user-defined functions.

This directive allows scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that is never used. The function call can be removed since the function has no side effects—it does not matter whether or not the call is made. Such optimization opportunities usually arise after other optimizations are performed and rarely occur in the original source text.

Although the directive can appear anywhere in a program unit, to be effective, it should be used before the named function is called. Use the directive if the compiler gives the advisory message "More optimization is possible if this function call has no side effects." If there are no arguments, the directive applies to all functions referenced (textually) after the directive.

Example 1:

```
C$DIR NO_SIDE_EFFECTS (F1,F2)
      ...
      X = Y * F1(5,Z) - W
      ...

*if the X= does not reach a use of X, the assignment
*statement may be removed
```

A function call with no side effects is invariant with respect to a loop, provided its arguments are loop invariant and the call may be moved out of the loop.

Example 2:

A function call may inhibit code motion. The directive is not applicable; the user must perform the optimization at the source level. (The source would have to be modified to add the directive.)

```
        DO I = 1,N
        ...
        Z = F3(A)

    *if F3 has no side effects and A is invariant, Z=can be
    *removed from the loop which may make Z loop invariant
        ...
        ENDDO
        ...
```

Equivalent code is

```
        T1 = F3(A)
        DO I = 1,N
        ...
        Z = T1
        ...
    ENDDO
```

Code motion moves the Z=.

C.2 Control Directives

The control directives control optimization, parallelization, and vectorization. The control directives are

- BEGIN_TASKS, NEXT_TASK, END_TASKS
- SCALAR
- FORCE_PARALLEL, FORCE_VECTOR
- PSTRIP, VSTRIP
- ROW_WISE
- SELECT
- SYNCH_PARALLEL
- UNROLL

The scope of a vectorization directive is the loop immediately following the directive; the scope does not, however, apply to loops nested therein. Some of the directives let you select loops or code sections to be parallelized rather than leaving the choice up to the compiler.

Certain combinations of directives are invalid when used within the same program unit and will cause the program unit to be rejected by the compiler. Table C-1 lists the invalid combinations.

Table C-1: Restrictions on Directive Use

The directive...	Cannot be used with...
FORCE_PARALLEL	SCALAR, SELECT, SYNCH_PARALLEL
FORCE_VECTOR	PSTRIP, SCALAR, SELECT, SYNCH_PARALLEL, VSTRIP
PSTRIP	FORCE_VECTOR, SCALAR
SCALAR	FORCE_PARALLEL, FORCE_VECTOR, PSTRIP, SELECT, SYNCH_PARALLEL, VSTRIP
SELECT	FORCE_PARALLEL, FORCE_VECTOR, SCALAR
SYNCH_PARALLEL	FORCE_PARALLEL, FORCE_VECTOR, SCALAR
VSTRIP	FORCE_VECTOR, SCALAR

C.2.1 Tasking Directives

The tasking directives let you specify a group of independent tasks for parallel execution. The tasking directives are

- BEGIN_TASKS
- NEXT_TASK
- END_TASKS

The BEGIN_TASKS directive identifies the beginning of the task group; the NEXT_TASK directive identifies each individual task in the group; and the END_TASKS directive terminates the task group. The following code illustrates the use of these directives:

```
C$DIR BEGIN_TASKS
  statement
  ...
C$DIR NEXT_TASK
  statement
  ...
C$DIR NEXT_TASK
  statement
  ...
C$DIR END_TASKS
```

The preceding example is equivalent to the following loop:

```
C$DIR FORCE_PARALLEL
      DO 100 I = 1,3
        GOTO (10,20,30),I
 10       statement-1
        GOTO 100
 20       statement-2
        GOTO100
 30       statement-3
100   CONTINUE
```

A maximum of 255 tasks can be specified between a BEGIN_TASKS and an END_TASKS directive.

C.2.2 SCALAR Directive

The SCALAR directive prevents the DO loop that follows from being vectorized or parallelized. The format of this directive is

SCALAR

The body of the loop may still be vectorized or parallelized if an outer loop interchanges with the SCALAR loop. The SCALAR directive is useful when the iteration count of the loop is too low for the overhead involved in setting up vectorization, or when the numerical results must be the same as for a scalar loop. This directive can also be used to prevent loop interchange, which may not choose the best loop to interchange when it cannot deduce the iteration counts of the loops involved.

It is possible for the results of a vectorized loop to differ from its scalar equivalent. For example, floating-point sum and product reduction operators may give different answers because the underlying hardware does not process the operands in sequential order.

In the following example, the compiler normally interchanges the I loop with the J loop so that elements of A, B, and C are accessed contiguously. The SCALAR directive ensures that the loop of greater iteration count is retained as the innermost loop.

```
C$DIR  SCALAR
       DO 10 I = 1,N        (where N = 2)
       DO 10 J = 1,M        (where M = 1000)
10     A(I,J) = B(I,J) + C(I,J)
```

In the following example, neither iteration count is sufficient to warrant vectorizing the loops.

```
C$DIR   SCALAR
        DO 10 I = 1,N       ! (where N = 2)
C$DIR   SCALAR
        DO 10 J = 1,M       ! (where M = 2)
10      A(I,J) = B(I,J) + C(I,J)
```

C.2.3 Force Directives

The force directives tell the compiler that the following loop is to be either parallelized or vectorized regardless of apparent recurrences or loop dependencies. The force directives are

- FORCE_PARALLEL
- FORCE_VECTOR

The FORCE_PARALLEL directive tells the compiler that the iterations of the following loop are independent and that the loop should be parallelized. The FORCE_VECTOR directive also implies that the iterations of the following loop are independent but tells the compiler to vectorize, rather than parallelize, the loop. If both FORCE_PARALLEL and FORCE_VECTOR are specified for the same loop, the loop is vectorized and the resulting strip mine loop is parallelized.

Both FORCE_PARALLEL and FORCE_VECTOR ignore any dependencies between iterations that the compiler may have located. Also, even though you use these directives on a loop, you may not get the desired code transformation if the compiler cannot generate the requested code; scalar recurrences usually cause this sort of problem.

It is possible to use a FORCE_VECTOR directive with a loop that was fully vectorized and get incorrect answers since the directive causes the compiler to ignore dependencies.

A loop can be executed in the following ways:

If you specify...	Then the loop is processed as...
scalar	serial
vector	vector but not parallel
parallel	parallel but not vector
parallel, vector	parallel outer strip and vector inner strip

Both loops that contain calls and loops that do not contain calls may be parallelized with the FORCE_PARALLEL directive. The FORCE_VECTOR directive should only be used with fully vectorizable loops. Neither directive can be used with the SCALAR directive or with the NO_RECURRENCE directive or an error condition results.

The FORCE_VECTOR directive can only be used on innermost loops. The FORCE_PARALLEL directive may be used on any parallelizable loop that does not contain a loop preceded by the FORCE_PARALLEL directive. The FORCE_PARALLEL directive is effective only if the -O3 compiler option is specified.

C.2.4 Strip-Mine Directives

The strip-mine directives allow you to control the strip-mine length for the associated parallel or vector loop. The strip-mine directives are

- PSTRIP
- VSTRIP

If you do not specify one of the strip-mine directives, the compiler selects a default value. Table C-2 shows the maximum strip mine lengths that the compiler uses when you specify the -O3 and -ep compiler options. In the table, *n* is the actual vector length and *ep* is the number of processors specified in the -ep option.

Table C-2: Maximum Strip Mine Lengths

Processors (-ep)	Parallel Strip Length	Vector Strip Length
1	1	128
more than 1	max(n/(2*ep),1)	max(min((n+ep-1)/ep,128),8)

The actual strip length per iteration is the smaller of: the number of iterations remaining to be processed or the maximum length of a strip from the table.

Examples:

The following examples show the maximum and actual vector strip lengths when the system includes four processors (-ep=4).

Trip Count	Maximum Strip Length	Actual Strip Length(s)
2	8	2
514	128	128, 128, 128, 128, 2 (for the 5 iterations)

If you do not specify the -ep directive, the compiler selects a default value for strip mining that is appropriate for the architecture of the machine on which you are compiling. The PSTRIP or the VSTRIP directive overrides the default values shown in the table.

C.2.4.1 PSTRIP Directive

The PSTRIP directive tells the compiler that the parallel loop associated with this directive should be strip-mined using the specified length. The format of this directive is

PSTRIP (*n*)

The value *n* is the strip mine length; if this directive is not used, the compiler selects a default value according to an internal algorithm. The PSTRIP directive cannot be used with vector loops.

C.2.4.2 VSTRIP Directive

The VSTRIP directive tells the compiler that the following vector loop should be strip mined with the specified length. The format of this directive is

VSTRIP (*n*)

The value *n* is the strip mine length and must be less than or equal to 128. The VSTRIP directive lets you reduce strip-mine length to create more iterations of the strip-mine loop so that it can be effectively parallelized.

C.2.5 ROW_WISE Directive

The ROW_WISE directive tells the compiler that the designated array names have their dimensions reversed. Reversing the order of subscripts can assist in vectorization. The format of this directive is

ROW_WISE (*array_name* [,*array_name*...])

The following cautions apply to the use of the ROW_WISE directive:

- Implicit array I/O, such as READ(5,*)A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears reversed when viewed in the debugger.
- If the ROW_WISE directive is applied to a dummy argument, the actual argument must also appear in a ROW_WISE directive within the caller. The compiler cannot detect this situation.

The following example illustrates the type of situation in which use of the ROW_WISE directive can improve performance of a program.

```
C$DIR ROW_WISE (A)
DIMENSION A(4,1000)
DO I = 1,4
  DO J = 1,1000
    A(I,J) = 0
  ENDDO
ENDDO
```

Although the preceding example vectorizes, performance is slowed because the array is being accessed via non-contiguous memory (FORTRAN stores arrays in column major order). If, however, the code segment in the preceding example were preceded by the directive C$DIR ROW_WISE (A), it would be interpreted by the compiler as follows:

```
DIMENSION A(1000,4)
DO I = 1,4
  DO J = 1,1000
    A(J,I) = 0
  ENDDO
ENDDO
```

The array is now being accessed from contiguous memory, thus increasing the execution speed.

C.2.6 SELECT Directive

The SELECT directive causes the compiler to generate multiple versions of a loop and to select, at runtime, which version to execute based on a specified trip count. This directive has the form

SELECT (*type1, type2* [*,count*])

The parameters *type1* and *type2* indicate the type of loop to be generated and can have the following values:

Loop Type	Explanation
S or s	Generate scalar loop
V or v	Generate vector loop

If the actual loop iteration count as determined at runtime is less than or equal to the iteration count specified in the directive, then the *type1* version of the loop is executed; otherwise, the *type2* version is executed. The iteration count is optional. If given, it must be a positive integer greater than 0. If no iteration count is given, the compiler assumes a default value of 4.

C.2.7 SYNCH_PARALLEL Directive

The SYNCH_PARALLEL directive tells the compiler that the following loop should be executed in parallel even though it requires synchronization that might result in less than full efficiency. The format of this directive is

SYNCH_PARALLEL

This directive is effective only if the -O3 option is specified on the compiler command line.

The loop in the following example might run faster in a machine with four processors than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

```
C$DIR SYNCH_PARALLEL
      DO I = 1,32
        IF (A(I).LT.0) THEN
          A(I) + A(I-1) + B(I)
          D(I) = E(I)*F(I)
        ENDIF
      ENDDO
```

C.2.8 UNROLL Directive

The UNROLL directive reduces loop overhead by replicating the body of the loop that follows. Unrolling is performed on both scalar and vector loops. The format of this directive is

UNROLL

To be eligible for unrolling, a loop must contain no internal branching and must have an iteration count that can be determined by the compiler. The compiler unrolls a loop completely only if its iteration count is known to be less than 5; otherwise, partial unrolling is performed.

In order for this directive to take effect, optimization level -O3 or -O2 must be specified on the compiler command line.

D

System Limits

This appendix lists the maximum sizes for the various elements in a CONVEX FORTRAN program.

The maximum...	**Is...**
Statement length	13,200 characters
Hollerith length	2000 characters
String length	65,535 characters
Identifier length	42 characters
Cross-reference identifier	31 characters
File name length	200 characters
INCLUDE nesting	127
Number of files in a program	127

Additionally, the total space used by all dimensioned arrays in a program must not exceed the memory space of the system architecture.

E

ASCII Character Set

This appendix lists the American Standard Code for Information Interchange (ASCII) with each character equivalent in hexadecimal and octal values. The FORTRAN character set is a subset of the ASCII character set; i.e., although not all of the ASCII characters are FORTRAN characters, all of the FORTRAN characters are included in the ASCII character set.

Table E-1: ASCII Character Set

Hex Value	Oct Value	Char	Hex Value	Oct Value	Char	Hex Value	Oct Value	Char
00	000	NUL	20	040	SP	40	100	@
01	001	SOH	21	041	!	41	101	A
02	002	STX	22	042	”	42	102	B
03	003	ETX	23	043	#	43	103	C
04	004	EOT	24	044	$	44	104	D
05	005	ENQ	25	045	%	45	105	E
06	006	ACK	26	046	&	46	106	F
07	007	BEL	27	047	’	47	107	G
08	010	BS	28	050	(	48	110	H
09	011	HT	29	051	)	49	111	I
0A	012	LF	2A	052	*	4A	112	J
0B	013	VT	2B	053	+	4B	113	K
0C	014	FF	2C	054	,	4C	114	L
0D	015	CR	2D	055	-	4D	115	M
0E	016	SO	2E	056	.	4E	116	N
0F	017	SI	2F	057	/	4F	117	O
10	020	DLE	30	060	0	50	120	P
11	021	DC1	31	061	1	51	121	Q
12	022	DC2	32	062	2	52	122	R
13	023	DC3	33	063	3	53	123	S
14	024	DC4	34	064	4	54	124	T
15	025	NAK	35	065	5	55	125	U
16	026	SYN	36	066	6	56	126	V
17	027	ETB	37	067	7	57	127	W
18	030	CAN	38	070	8	58	130	X
19	031	EM	39	071	9	59	131	Y
1A	032	SUB	3A	072	:	5A	132	Z
1B	033	ESC	3B	073	;	5B	133	[
1C	034	FS	3C	074	<	5C	134	\
1D	035	GS	3D	075	=	5D	135	]
1E	036	RS	3E	076	>	5E	136	^
1F	037	US	3F	077	?	5F	137	_

Table E-1: ASCII Character Set (continued)

Hex Value	Oct Value	Char	Hex Value	Oct Value	Char	Hex Value	Oct Value	Char
60	140	‘	6B	153	k	76	166	v
61	141	a	6C	154	l	77	167	w
62	142	b	6D	155	m	78	170	x
63	143	c	6E	156	n	79	171	y
64	144	d	6F	157	o	7A	172	z
65	145	e	70	160	p	7B	173	{
66	146	f	71	161	q	7C	174	|
67	147	g	72	162	r	7D	175	}
68	150	h	73	163	s	7E	176	~
69	151	i	74	164	t	7F	177	DEL
6A	152	j	75	165	u			

F

FORTRAN-66 Compatibility

The CONVEX FORTRAN compiler adheres to the American National Standard FORTRAN-77, X3.9-1978, ISO 1539-1980(E), i.e., the default language interpretations are FORTRAN-77. The compiler can, however, compile FORTRAN-66 programs. There are five incompatibilities between American National Standard FORTRAN-77 and FORTRAN-66, X3.9-1966.

- EXTERNAL statement
- DO loop minimum iteration count
- OPEN statement BLANK keyword default
- OPEN statement STATUS keyword default
- X format edit descriptor

The first two incompatibilities are interpreted by the compiler; the rest are interpreted by the runtime system. If your program uses the OPEN statement and you want FORTRAN-66 interpretation rules at runtime, either include a call to *ioinit* in your main program or include the library I66 in your link by using the option -lI66 on the *fc* command line. The X format edit descriptor use must be modified.

F.1 Compiling FORTRAN-66 Programs

To compile a FORTRAN-66 program, you can modify the program, transforming it into a FORTRAN-77 program, and/or use the -F66 option.

1. Use *grep* to identify OPEN statements in which a STATUS keyword is to be added, EXTERNAL statements that must be changed to INTRINSIC statements, and FORMAT statements using the X edit descriptor.

2. Use the -F66 option or OPTIONS statement to select FORTRAN-66 language interpretations. The -F66 option allows for the interpretation of EXTERNAL statements, DO loop minimum iteration counts, and BLANK and STATUS keyword defaults in OPEN. It does not affect the X format edit descriptor. To avoid including the -F66 option in the *fc* command each time, use the *alias* command as follows:

   ```
   alias fc fc -F66
   ```

   You can include this format in your *.cshrc* file, with the $ parameter representing specified files. (When you use the *alias* command, you must be using C shell not the Bourne shell, as it has no *alias* command.)

F.2 EXTERNAL Statement

In FORTRAN-66, the EXTERNAL statement specifies that a symbolic name is the name of either a user-defined external procedure or a FORTRAN-supplied function. In FORTRAN-77, two statements accomplish this function:

- The INTRINSIC statement specifies that the procedure is a FORTRAN-supplied intrinsic procedure, such as SQRT.
- The EXTERNAL statement specifies that the procedure is user-supplied.

Because of the exact specification of these two procedures, you cannot modify the EXTERNAL statements in your program so that the same source program works with both FORTRAN-77 and FORTRAN-66. You must substitute an equivalent statement to include the changes:

FORTRAN-66	FORTRAN-77
EXTERNAL USER	EXTERNAL USER (no change)
EXTERNAL SQRT	INTRINSIC SQRT
EXTERNAL *SQRT	EXTERNAL SQRT (where SQRT is a user function not the intrinsic for the square root)

F.3 DO Loop Minimum Iteration Count

In FORTRAN-66 the body of a DO loop is always executed; in FORTRAN-77 the body of the DO loop is not executed if the end condition of the loop is already satisfied when the DO statement is executed. To run a FORTRAN-66 program with the FORTRAN-77 compiler, you can either use the -F66 option, or modify the DO statements in the program to ensure a minimum loop count of 1; e.g., in FORTRAN-77, the loop

```
DO  20  J=INIT,LAST
```

is not executed if INIT > LAST, but is executed once in FORTRAN-66. If this DO statement occurs in a FORTRAN-66 program, its equivalent FORTRAN-77 statement is:

```
DO  20  J=INIT,MAX(INIT,LAST)
```

F.4 OPEN Statement Keywords

While FORTRAN-66 does not contain an OPEN statement, it does allow for many implementations based on FORTRAN-66 which contain an OPEN statement. Both the BLANK and STATUS keywords in OPEN for FORTRAN-77 differ from the implementations which are utilized under FORTRAN-66.

F.4.1 BLANK keyword

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D, E, F, G, I, O, and Z field descriptors. In FORTRAN-77, the OPEN statement BLANK keyword defaults to BLANK='NULL' (which means that blanks in numeric fields are ignored). The FORTRAN-66 interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

When a logical unit is opened without an explicit OPEN statement, CONVEX FORTRAN and FORTRAN-66 both provide a default equivalent to BLANK='ZERO'.

The use of BLANK='NULL' causes embedded and trailing blanks to be ignored and the value converted as if the nonblank characters were right-justified in the field. However, the use of BLANK='ZERO' causes embedded and trailing blanks to be treated as zeros.

If your program treats blanks in numeric input fields as zeros, and you do not want to use -lI66 or *ioinit*, include BLANK='ZERO' in the OPEN statement.

F.4.2 STATUS Keyword

The OPEN statement STATUS keyword in FORTRAN-77 specifies the initial status of the file (OLD, NEW, SCRATCH, or UNKNOWN); its default value is UNKNOWN. In FORTRAN-66, where STATUS is called TYPE, the default value is NEW.

If your program assumes that the default value for TYPE is NEW and you do not want to use -lI66 or *ioinit*, put STATUS = 'NEW' in the OPEN statement.

F.5 X Format Edit Descriptor

The FORTRAN-66 implementation of the X format edit descriptor writes blanks to and may extend the output record. The FORTRAN-77 version does not modify character positions that are skipped and does not, as a result, affect the length of the output record.

F.5.1 Format Code Separators

Formats with no format code separators are supported.

G

Cray FORTRAN Compatibility

To facilitate porting code written for the Cray compiler, the *-cfc* option can be specified on the *fc* command line. When used, this option changes certain aspects of the CONVEX FORTRAN compiler as follows:

- The default data types are as follows:

Type	Default Length
integer	INTEGER*8
real	REAL*8
complex	COMPLEX*16
logical	LOGICAL*8

- Constants are stored in the default INTEGER or REAL type.
- Constants written in exponential form with an E (e.g., 1.23E4) are stored in REAL*8 format.
- Intrinsics that work with default integer and single-precision types work with Cray default types (8-byte quantities).
- LOGICAL*2 and *4, INTEGER*4, and REAL*4 are treated as LOGICAL*8, INTEGER*8, and REAL*8.

The following Cray features are not supported:

- Double-precision (REAL*16) data types; that is, constants written with a D in the exponent or objects declared double-precision.
- Cray special features such as intrinsics and directives.

H

VAX FORTRAN Compatibility

This Appendix describes compatibility between VAX FORTRAN and CONVEX FORTRAN. To facilitate porting code written for the VAX compiler, certain VAX features are supported as described below.

H.1 Supported Features

CONVEX FORTRAN supports the VAX INCLUDE statement, but VMS file names are not supported. CONVEX FORTRAN supports the following VAX FORTRAN features only when the *-vfc* option is specified on the *fc* command line.

- The alternate form (without parentheses) of the PARAMETER statement with only one constant specified.
- The 'r form of the record specifier.
- Octal constants in the form "nn, where nn is a string of octal digits.
- The RECL= specifier used in the OPEN and INQUIRE statements returns the number of VAX words rather than bytes for unformatted files.
- Default file names in the form "FOR0nn.DAT", where the number *nn* corresponds to unit number *nn*.
- VAX FORTRAN records.

The organization and structure of VAX FORTRAN records is discussed briefly at the end of this appendix.

H.2 Unsupported Features

CONVEX FORTRAN does not support the following VAX FORTRAN features:

- REAL*16 data type
- %DESCR
- RMS calls
- The VOLATILE statement
- The zccc...c form of hexadecimal constants
- Interactive display of NAMELIST group and values or end-of-line comments (!) in the NAMELIST input data.

- Byte ordering with respect to characters and parameter passing is not VAX compatible.
- Logical values differ in the two compilers. On CONVEX, a logical value is true if it is all 1. On VAX, it is true in a test if the low-order bit is 1, i.e., IF(A) is equivalent to IF(A .AND. 1).
- Numerical differences exist owing to the accuracy of CONVEX floating-point representation and the rounding method used. See the *CONVEX Architecture Handbook* for further information.
- Hollerith constants can be used where a character value is expected.
- Unlike VAX FORTRAN, CONVEX FORTRAN does not allow you to call a function as a subroutine.
- CONVEX FORTRAN does not support radix 50 constants.
- CONVEX FORTRAN does not allow the variable on the left-hand side of a character assignment statement to appear on the right-hand side. VAX FORTRAN allows the construct as an extension.
- CONVEX does not allow the MOD function to be defined for a zero denominator.
- The ANSI standard prohibits modifying an argument within a subroutine if the subroutine was called with a constant in the argument list. The CONVEX FORTRAN compiler enforces this rule.
- CONVEX does not support DO statements of the form:

  DO 714 J=1, 100 WHILE (Q.NE.Z)

The product does not support certain VAX FORTRAN I/O extensions:

- REWRITE, DELETE, and UNLOCK statements
- Indexed I/O (key-indexed files)
- File sharing
- DEFINEFILE statement
- OPEN keywords (PRINT and SUBMIT values for DISPOSE; USEROPEN; INITIALSIZE; EXTENDSIZE; BUFFERCOUNT; SEGMENTED for RECORDTYPE; and ORGANIZATION).
- CLOSE keywords (PRINT and SUBMIT values for STATUS)
- ASCII null as a carriage control character

The internal format of variable-length type records of VAX FORTRAN and CONVEX FORTRAN differ when RECORDTYPE=VARIABLE.

The following table shows how the VAX and CONVEX versions of the STOP message differ.

Statement	CONVEX Message	VAX Message
STOP STOP 4 STOP 'here'	STOP: STOP: 4 STOP: here	FORTRAN STOP 4 here

Integer overflow traps are turned off by default in *fc*, whereas VAX FORTRAN enables them by default. The main reasons for this are:

- Overflow is turned off in C, and many users mix C and FORTRAN code.
- It is difficult to optimize integer expressions if integer overflow is turned on because most addresses are negative integers near overflow on a CONVEX machine.

When you use the H descriptor on input, the first character transferred appears immediately after the letter H. The characters that are in the field descriptor before input are replaced (overwritten) by the new input characters.

H.3 Miscellaneous Differences

The following miscellaneous differences exist between VAX FORTRAN and CONVEX FORTRAN:

- Unit numbers in VAX FORTRAN range from 0 to 99 and on CONVEX FORTRAN from 0 to 255.
- VAX FORTRAN supports the use of both Hollerith and apostrophe edit descriptors during formatted input. CONVEX FORTRAN allows Hollerith but does not allow apostrophe edit descriptors.
- In VAX FORTRAN, 'cc...c'O octal constants are of type integer; in CONVEX FORTRAN, the 'cc...c' form of octal constants is typeless.
- If invalid data is encountered on a VAX FORTRAN READ statement, all variables on the *iolist* are assigned except those corresponding to the bad data. If the same error occurs in CONVVEX FORTRAN, the READ statement terminates at once and the remaining variables in the *iolist* are unchanged.

H.4 VAX FORTRAN Records

A VAX FORTRAN record consists of one or more fields. The fields within a record are defined by means of a structure declaration that defines the field names, types of data within fields, and order and alignment of fields within a record.

H.4.1 Structure Declaration

A structure declaration is bounded by STRUCTURE and END STRUCTURE statements and contains one or more field declarations. The order in which the field declarations occur determines the order of fields within the structure. At least one field declaration must be specified or an error condition occurs.

A structure declaration has the following format:

> **STRUCTURE** */structure-name/*
> *field declaration*
> ...
> **END STRUCTURE**

A structure declaration does not create a variable. A variable is created by a RECORD statement containing the name of a previously-declared structure. The format of a record statement is as follows:

> **RECORD** */structure-name/record-namelist*

where *structure-name* is the name of a previously declared structure and *record-namelist* is a list of variable names, array names, or array declarations separated by commas.

H.4.2 Field Declaration

A field declaration can be any combination of the following: a typed data declaration, a substructure declaration, or a union declaration.

A typed data declaration is the same as a normal FORTRAN type statement. As with FORTRAN typed data statements, field declarations may contain initializers. The name %FILL can be used in place of a field name to create space in the structure for padding, although this space cannot be initialized.

A substructure declaration can be declared by a RECORD statement that creates an instance of a previously-declared structure.

A union declaration is bounded by UNION and END UNION statements and defines a data area that can be shared during program execution. A union declaration must contain at least two map declarations (see below) or an error condition occurs. A union declaration has the following format:

> **UNION**
> *map declaration*
> *map declaration*
> ...
> **END UNION**

A map declaration defines a unique group of fields and is bounded by MAP and END MAP statements. A map declaration must contain at least one field declaration or an error condition occurs. A map declaration has the following format:

> **MAP**
> *field declaration*
> ...
> **END MAP**

I

Sun FORTRAN Compatibility

To facilitate porting code written for the Sun FORTRAN compiler, the *-sfc* option can be specified on the *fc* command line. When used, this option changes certain aspects of the CONVEX FORTRAN compiler as follows:

- As in the C language, escape sequences using a backslash (\) are supported in character strings to define nonprintable characters. The following table lists the supported sequences.

Character	Sequence
newline	\n
tab	\t
form feed	\f
NUL	\0
single quote	\'
double quote	\"
backslash	\\

- The ampersand (&) character in the first nonblank column of a source line indicates that the line is a continuation, regardless of what is in column 6.

- Recursive subroutines and functions are allowed.

- The declarations AUTOMATIC and STATIC are supported.

The preceding features represent only a subset of Sun FORTRAN FORTRAN that has been made available in CONVEX FORTRAN.

J
Preprocessor

The compiler contains a preprocessor that was used in previous versions but is now optional. The preprocessor is retained for support purposes only and should not be used unless absolutely necessary. Certain language elements may not be available when the preprocessor is used. In addition, macros in Hollerith strings are not expanded and may cause problems with some programs.

J.1 Preprocessor Statements

Preprocessor statements begin with the # symbol and are syntax-independent of the compiler. You can continue long statements by entering a backslash at the end of the line to be continued.

J.1.1 #define Statement

The #define statement causes the preprocessor to replace subsequent instances of an *identifier* with a given *token-string*. This statement has the form:

#define *identifier token-string*
#define *identifier(identifier,..., identifier) token-string*

The *token string* in the definition replaces the *identifier*. The arguments in the call in the second form are *token strings* separated by commas. Commas within quoted strings or protected by parentheses do not separate arguments. The corresponding *token string* from the call replaces every *identifier* mentioned in the formal parameter list of the definition, and the number of formal and actual parameters must be the same. Text inside a string or a character is not replaced.

Blanks are significant in #define statements. In addition, the argument list must immediately follow the macro name.

J.1.2 #undef Statement:

The #undef statement causes the identifier preprocessor definition to be removed. This statement has the form:

#undef *identifier*

J.1.3 #include Statement

The #include statement replaces the line on which the statement appears by the contents of a specified file. This statement has the form:

#include "*filename*"

This statement searches for the specified file in the directory of the original source file, then along the paths specified in -I options on the command line, and finally along a standard search path. *#include* statements may be nested.

J.1.4 #if Statement

The #if statement checks whether a constant expression evaluates to nonzero, whether the identifier is currently defined in the preprocessor, or whether the identifier is currently undefined in the preprocessor. This statement has the forms:

#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*

These forms are followed by an arbitrary number of lines that may contain a control line *#else*. The last line must be *#endif*. If the condition is true, the lines between the *#else* and the *#endif* are ignored; if the checked condition is false, the lines between the *#if* and an *#else* (or the *#endif*, if no *#else* exists) are ignored. These constructions may be nested.

J.2 Preprocessor Options

Certain compiler command line options are associated exclusively with the preprocessor. These options, with the exception of *-fpp*, are being phased out. The options are as follows:

-D*name*[=*def*] Defines *name* to the preprocessor as if it had been specified in a *#define* statement. If no definition is given, the name is defined as 1.

-E Runs only the FORTRAN preprocessor on the named FORTRAN programs and sends the result to the standard output file.

-fpp Runs the FORTRAN preprocessor as the first step of compilation. If this option is not specified, the preprocessor is not used.

-I*dir* For #include files whose names do not begin with "/", the preprocessor searches first in the directory of the file argument, then in directories named in -I, then in directories on a standard list.

-U*name* Removes any initial definition of *name*. The only built-in names defined are "__LINE__", and "__FILE__".

J.3 Preprocessor Messages

During compilation, error messages can be generated by either the FORTRAN preprocessor, *fpp*, or by the FORTRAN compiler, *fc*. The preprocessor generates error messages and warning messages in the following format:

```
fpp: file: line_number: error message
```

The preprocessor error and warning messages are designed to be self-explanatory.

K

Problem Reporting

K.1 Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

You must know the name and version number of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command. The syntax for the command is

vers *filename*

where *filename* is the the full pathname of the program. If you don't know the full pathname of the program, type

which *program*

For more information on these commands, see *vers*(1) and *which*(1) in the *CONVEX UNIX Programmer's Manual*, Part I.

K.2 Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.
2. The name and version of the product involved. Use the *vers* command if you don't know the version number of the program or utility.
3. A short (1 line) summary of the problem.
4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.
5. The priority of the problem. You are shown a list of six levels from which to select.

the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report.*

The following figure is a sample *contact* session. User input is in bold lettering, and the system response is in constant-width lettering.

Figure K–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.14 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University (RETURN)
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed." Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

Index

$ descriptor lrm-8-16
: descriptor lrm-8-17

A
ACCEPT statement lrm-7-1, lrm-7-11
ACCESS keyword lrm-7-18
accessing files lrm-7-3
actual arguments lrm-9-1
adjustable arrays lrm-9-2
alternate return arguments lrm-9-5, lrm-9-9
ANSI standard formatting lrm-1-4
apostrophe descriptor lrm-8-5
arguments lrm-9-8, lrm-9-9
arguments, alternate return lrm-9-9
arguments, character lrm-9-3
arguments, dummy lrm-9-9
arithmetic expression, data type lrm-2-9
arithmetic expression, data types lrm-2-8
arithmetic expressions lrm-2-8
arithmetic IF statement lrm-6-3
arithmetic operators lrm-2-8
array arguments lrm-9-2
array declaration lrm-2-7
array storage lrm-2-8
array subscripts lrm-2-7
arrays lrm-2-7
ASCII character set lrm-E-1
ASSIGN statement lrm-5-3
assigned GOTO statement lrm-6-2
assignment statement lrm-5-1
ASSOCIATEVARIABLE keyword lrm-7-20
assumed-length character argument lrm-3-4
assumed-size arrays lrm-9-3
auxiliary input/output statements lrm-7-18

B
B descriptors lrm-8-12
BACKSPACE statement lrm-7-28
BEGIN_TASKS directive lrm-C-4
blank common storage lrm-3-1
BLANK keyword lrm-7-20, lrm-F-2
BLOCK DATA statement, definition lrm-10-1
BLOCK DATA statement in subprograms lrm-9-9
block IF statement lrm-6-4
blockdata subprogram lrm-10-1
BLOCKSIZE keyword lrm-7-20
BN descriptor lrm-8-12
built-in functions lrm-9-6
built-in functions %REF and %VAL lrm-9-6
BYTE lrm-2-1
BZ descriptor lrm-8-12

C
CALL statement lrm-9-10
carriage-control characters lrm-8-24
CARRIAGECONTROL keyword lrm-7-20
-*cfc* option lrm-G-1
character arguments lrm-9-3
character constants lrm-2-6
character conversions lrm-5-1
character descriptor A lrm-8-4
character equivalence lrm-3-5
character expressions lrm-2-10
CHARACTER FUNCTION statement lrm-9-9
character substrings lrm-2-11
CHARACTER type-declaration statements lrm-3-3
character-per-column formatting lrm-1-3
characters, carriage-control lrm-8-24
CLOSE statement lrm-7-25
coding statements lrm-1-1
colon descriptor lrm-8-17
comma field separator lrm-8-18
comment line lrm-1-2
common block storage lrm-3-1
common blocks lrm-3-6
COMMON statement lrm-3-1
COMMON statements lrm-2-6
compiler directives lrm-1-4, lrm-C-1
compiler options lrm-B-1
compiling FORTRAN-66 programs lrm-F-1
compiling programs lrm-B-1
COMPLEX lrm-2-1
complex descriptor lrm-8-4
COMPLEX*16 lrm-2-1
COMPLEX*16 constants lrm-2-4
COMPLEX*8 lrm-2-1, lrm-2-9
COMPLEX*8 constants lrm-2-4
computed GOTO statement lrm-6-2
concatenation lrm-2-10
constant expressions lrm-2-11
constants lrm-2-3
contact, reporting problems lrm-K-1
continuation indicator lrm-1-3
continuation line lrm-1-3
CONTINUE statement lrm-6-9
control statement lrm-6-1
conversion of data types lrm-2-2
COVUEshell lrm-1-7, lrm-7-18
Cray compatibility lrm-G-1

D
D descriptor lrm-8-9
D indicator lrm-1-4
DATA statement lrm-4-1
DATA statement form lrm-4-1
DATA statement, implied-DO lrm-4-2
data types lrm-2-1
data types, arithmetic expression lrm-2-8
data types, conversion lrm-2-2
data types, equivalenced lrm-3-4
data-type length specifiers lrm-3-3
-dc option lrm-1-4
debug statements lrm-1-4
DECODE statement lrm-7-16
default descriptor values lrm-8-17
DEFAULTFILE keyword lrm-7-21
descriptors lrm-8-3
dimension declarator lrm-2-7
DIMENSION statement lrm-3-4

direct access lrm-7-4
direct-access WRITE statements lrm-7-13
directives, compiler lrm-1-4, lrm-C-1
DISPOSE keyword lrm-7-21
DO loops, extended range lrm-6-8
DO statement lrm-6-8, lrm-6-9
DO statements lrm-6-6
DO WHILE statement lrm-6-8, lrm-6-9
dollar sign descriptor lrm-8-16
DOUBLE PRECISION lrm-2-1
double-Precision constants lrm-2-3
dummy argument lrm-9-1
dummy argument, NAMELIST statement lrm-3-8
dummy arguments lrm-9-1, lrm-9-8, lrm-9-9
dummy arguments, arrays lrm-9-2

E

E descriptor lrm-8-9
edit descriptors lrm-8-3
ENCODE statement lrm-7-15
ENCODE statement, example lrm-7-16
END DO statement lrm-6-9
END statement lrm-6-10
END statement in subprograms lrm-9-9
ENDFILE record lrm-7-2
ENDFILE statement lrm-7-28
end-of-file specifier lrm-7-7
END_TASKS directive lrm-C-4
ENTRY statement lrm-9-11
EQUIVALENCE statement lrm-3-4
EQUIVALENCE statements lrm-2-6
equivalencing arrays lrm-3-5
ERR keyword lrm-7-21
error reporting lrm-K-1
error specifier lrm-7-7
executable program lrm-1-1
expressions lrm-2-8
expressions, constant lrm-2-11
external READ statements lrm-7-9
EXTERNAL statement lrm-3-9

F

F descriptor lrm-8-8
fc command line lrm-B-1
field descriptors lrm-8-3
field separators, external lrm-8-18
FILE keyword lrm-7-21
file positioning lrm-8-3
file-positioning statements lrm-7-28
files lrm-7-2
files, accessing lrm-7-3
FIND statment lrm-7-17
floating-point representation, IEEE lrm-B-3
force directives lrm-C-5
FORCE_PARALLEL directive lrm-C-5
FORCE_VECTOR directive lrm-C-5
FORM keyword lrm-7-22
format code separators lrm-F-3
FORMAT control lrm-8-2
format specifications lrm-8-1
format specifier lrm-7-6
FORMAT statement lrm-8-1
formats, variable lrm-8-19
formatted records lrm-7-2
formatting, list-directed lrm-8-20
FORTRAN character set lrm-1-1, lrm-E-1
FORTRAN-66 compatibility lrm-F-1
FORTRAN-77 formatting lrm-1-4
fpp lrm-J-1
FUNCTION statement lrm-9-9
function subprograms lrm-9-9
functions lrm-9-5

G

G descriptor lrm-8-10
generic and intrinsic functions, table lrm-A-1
GOTO statement lrm-6-1

H

H descriptor lrm-8-5
hexadecimal constants lrm-2-4, lrm-2-5
Hollerith constants lrm-2-5

I

I descriptor lrm-8-6
IEEE floating-point representation lrm-B-3
IF statements lrm-6-3
IF THEN statement lrm-6-4
IMPLICIT NONE statement lrm-3-2
IMPLICIT statement lrm-3-2
implied-DO list lrm-7-5
#include statement lrm-1-6
INCLUDE statement lrm-1-6
Inf message lrm-8-9, lrm-8-10, lrm-8-11
initial line lrm-1-3
input, list-directed lrm-8-20
input/output lists lrm-7-4
input/output statements lrm-7-1
input/output statements, auxiliary lrm-7-18
INQUIRE statement lrm-7-25
integer constants lrm-2-3
integer descriptor lrm-8-4
INTEGER*1 lrm-2-1
INTEGER*2 lrm-2-1
INTEGER*4 lrm-2-1
INTEGER*8 lrm-2-1
internal direct-access WRITE statements lrm-7-14
internal files lrm-7-2
internal READ statements lrm-7-11
intrinsic functions lrm-9-5
INTRINSIC statement lrm-3-9
invoking the compiler lrm-B-1
I/O statement format lrm-7-4
I/O statements lrm-7-1
IOSTAT keyword lrm-7-22

K
keywords, OPEN statement lrm-7-18

L
L descriptor lrm-8-5
limits, system lrm-D-1
list-directed character input lrm-8-20
list-directed complex input lrm-8-20
list-directed formatting lrm-8-20
list-directed input lrm-8-20
list-directed, null value lrm-8-21
list-directed output lrm-8-23
list-directed, slashes lrm-8-21
%LOC built-in function lrm-9-7
logical constants lrm-2-6
logical descriptor lrm-8-4
logical elements lrm-2-10
logical entities lrm-2-9
logical expressions lrm-2-10
logical IF statement lrm-6-3
logical operator .XOR. lrm-2-10
LOGICAL*1 lrm-2-1
LOGICAL*2 lrm-2-1
LOGICAL*4 lrm-2-1
LOGICAL*8 lrm-2-1

M
main program lrm-1-1, lrm-3-7
MAXREC keyword lrm-7-22
MAX_TRIPS directive lrm-C-1
multiple statements lrm-1-4

N
NAME keyword lrm-7-21
namelist input lrm-8-21
namelist specifier lrm-7-8
NAMELIST statement lrm-3-8
namelist-directed formatting lrm-8-21
namelist-directed output lrm-8-23
NaN message lrm-8-9, lrm-8-10, lrm-8-11
nested block IF statement lrm-6-6
nested DO loops lrm-6-7
NEXT_TASK directive lrm-C-4
NML keyword lrm-7-8
nonrepeatable descriptors lrm-8-2
NO_RECURRENCE directive lrm-C-1
NO_SIDE_EFFECTS directive lrm-C-2
NOSPANBLOCKS keyword lrm-7-23
numeric type-declaration statements lrm-3-3

O
O descriptor lrm-8-7
octal constants lrm-2-4, lrm-2-5
OPEN statement lrm-7-18
OPEN statement keywords lrm-F-2
operator precedence lrm-2-8
optimization directives lrm-C-3
options, compiler lrm-B-1
OPTIONS statement lrm-1-5

P
P descriptor lrm-8-13
parallelization directives lrm-C-3
PARAMETER statement lrm-3-6
PARAMETER statement, alternate lrm-3-7
PAUSE statement lrm-6-10
precedence, operator lrm-2-8
preprocessor lrm-J-1
PRINT statement lrm-7-1, lrm-7-14
procedures as dummy arguments lrm-9-4
program elements lrm-1-1
program, executable lrm-1-1
program, main lrm-1-1, lrm-3-7
PROGRAM statement lrm-1-1, lrm-3-7
PROGRAM statement in subprograms lrm-9-9
program unit lrm-1-1
PSTRIP directive lrm-C-7

Q
Q descriptor lrm-8-16

R
R descriptor lrm-8-14
READ statement lrm-7-1, lrm-7-8
READ statement, direct lrm-7-10
READ statement, external lrm-7-9, lrm-7-10
READ statement, internal lrm-7-11
READ statement, sequential lrm-7-9
READONLY keyword lrm-7-23
REAL lrm-2-1
real constants lrm-2-3
real descriptor lrm-8-4
REAL*4 lrm-2-1
REAL*4 Constants lrm-2-3
REAL*8 lrm-2-1, lrm-2-9
REAL*8 constants lrm-2-3
RECL keyword lrm-7-23
record specifier lrm-7-6
records lrm-7-1
RECORDSIZE keyword lrm-7-23
RECORDTYPE keyword lrm-7-23
%REF built-in function lrm-9-6
relational expressions lrm-2-9
repeatable descriptors lrm-8-2, lrm-8-3
reporting problems lrm-K-1
RETURN, alternate lrm-9-10
RETURN statement lrm-6-10, lrm-9-11, lrm-9-12
RETURN statement, definition lrm-9-12
RETURN statement in subprograms lrm-9-9
REWIND statement lrm-7-28
Rop message lrm-8-9, lrm-8-10, lrm-8-11
ROW-WISE directive lrm-C-7
runtime formats lrm-8-19

S
S descriptor lrm-8-13
SAVE statement lrm-3-9
SCALAR directive lrm-C-5

scale factor lrm-8-13
SELECT directive lrm-C-8
sequential access lrm-7-3
sequential READ statements lrm-7-9
sequential WRITE statement, unformatted lrm-7-13
sequential-access WRITE statements lrm-7-12
-sfc option lrm-I-1
slash descriptor lrm-8-17
SP descriptor lrm-8-13
specification statements lrm-3-1
specifiers lrm-7-5
SS descriptor lrm-8-13
statement field lrm-1-4
statement function reference lrm-9-8
statement functions lrm-9-7
statement label lrm-1-3
statement label assignment lrm-5-3
statements, executable lrm-1-2
statements, multiple lrm-1-4
statements, nonexecutable lrm-1-2
STATUS keyword lrm-7-24, lrm-F-3
status specifier lrm-7-7
STOP statement lrm-6-10
storage, array lrm-2-8
strip mining, parallel lrm-C-6
strip-mine directives lrm-C-6
SU descriptor lrm-8-14
subprogram lrm-1-1
subprograms lrm-9-1
SUBROUTINE statement in subprograms lrm-9-9
subroutine subprograms lrm-9-10
Sun FORTRAN compatibility lrm-I-1
symbolic Names lrm-2-1
SYNCH_PARALLEL directive lrm-C-8
system limits lrm-D-1

T

T descriptors lrm-8-15
tab character lrm-1-4
tab-key formatting lrm-1-4
tasking directives lrm-C-3, lrm-C-4
TL descriptor lrm-8-15
TR descriptor lrm-8-15
trouble reports lrm-K-1
TYPE keyword lrm-7-24
TYPE statement lrm-7-1, lrm-7-14
type-declaration statements lrm-3-2

U

unconditional GOTO statement lrm-6-1
unformatted records lrm-7-2
UNIT keyword lrm-7-24
unit specifier lrm-7-6
units lrm-7-3
UNROLL directive lrm-C-9

V

%VAL built-in function lrm-9-6
variable formats lrm-8-19
variables lrm-2-6
VAX FORTRAN compatibility lrm-H-1
VAX FORTRAN records lrm-H-3
vectorization directives lrm-C-3
vers command lrm-K-1
version of software, how to find lrm-K-1
-vfc option lrm-H-1
VMS FORTRAN compatibility lrm-H-1
VSTRIP directive lrm-C-7

W

which lrm-K-1
WRITE statement lrm-7-1, lrm-7-12, lrm-8-23
WRITE statement, list-directed output formats lrm-8-23
WRITE statements, direct access lrm-7-13
WRITE statements, internal lrm-7-14

X

X descriptor lrm-8-14
X format edit descriptor lrm-F-3

Z

Z descriptor lrm-8-7

CONVEX

Software Documentation

Index Enhancements

So that we can continue to provide better indexing in CONVEX documentation, please keep track of the words or phrases you look up in an index, but don't find. Then, list under which index entry you ultimately found the information you were seeking. You can mail one of these postage-paid forms to the CONVEX Software Documentation Department monthly, or you can submit the information to the Technical Assistance Center in the form of a bug report. You can get more forms by writing to CONVEX at the address below, or by calling us. You can also photocopy this form and mail it back in an envelope. Thank you for helping us to serve you better.

Name:____________________ **Company:**____________________

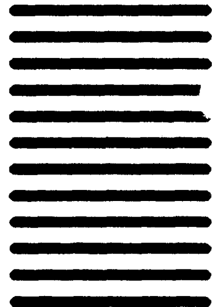
Phone:____________________ **Date:**____________________

Manual Title/Rev. No.	Looked Up This Word	Found Information Under This Word

CONVEX Computer Corporation — P.O. Box 833851, Richardson, TX 75083-3851 (214)952-0200

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)

CONVEX FORTRAN Language Reference Manual
Document No. 720-000050-203, Seventh Edition

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us in our efforts to generate quality customer documentation. (Please list the page number for questions and comments):

___

___

___

___

___

___

___

___

___

___

___

From:

Name____________________ Title____________________

Company____________________ Date____________________

Address and Phone No.________________________________

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Location	Phone Number
In Texas	(214)952-0379
Other continental locations	1(800)952-0379
Outside continental U.S.	Contact local CONVEX office

Direct Mail Orders to:

CONVEX Computer Corporation
Customer Service
Educational Department
P.O. Box 833851
Richardson, Texas 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)