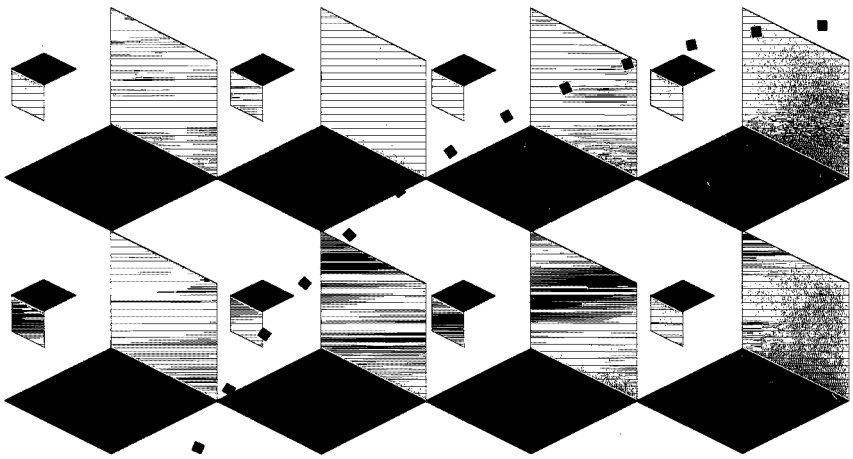


CONVEX

Fortran Language Reference

Eleventh Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

Fortran
Language Reference

Order No. DSW-037

Eleventh Edition
October 1994

CONVEX Press
Richardson, Texas
United States of America

Fortran
Language Reference

Order No. DSW-037

Printed in the United States of America

Revision information for Fortran Language Reference

Edition	Document No.	Description
Eleventh	720-002230-011	Released with CONVEX Fortran software V9.1, October 1994.
X11.0.0.2	720-002230-010	Released with CONVEX Fortran software V9.0.0.2, June 1994.
X11.0.0.1	720-002230-009	Released with CONVEX Fortran software V9.0.0.1, March 1994.
Tenth	720-002230-007	Released with CONVEX Fortran software V8.0, November 1992.
Ninth	720-002230-004	Released with CONVEX Fortran V7.0.0.1, August 1991.
Eighth Rev. 2	720-002230-003	Released with CONVEX Fortran V6.1.0.1, October 1990.
Eighth Rev. 1	720-002230-001	Released with CONVEX Fortran V6.0, March 1990.
Eighth	720-000050-204	Released with CONVEX Fortran V5.1, May 1989.
Seventh	720-000050-203	Released with CONVEX Fortran V5.0, November 1988.
Sixth Rev. 1	720-000050-202	Released with CONVEX Fortran V4.1, May 1988.
Sixth	720-000050-201	Released with CONVEX Fortran V4.0, November 1987.
Fifth	720-000050-200	Released with CONVEX Fortran V3.0, May 1987.
Fourth	720-000099-000	Released with CONVEX Fortran V2.2, September 1986.
Third	720-000150-000	Released with CONVEX Fortran V2.0, April 1986.
Second	720-000150-000	Released with CONVEX Fortran V1.7, September 1985.
First	720-000150-100	Released with CONVEX Fortran V1.0, February 1985. First release of the language reference.

Contents

Figures

Tables

- Chapter 5 discusses specification statements.
- Chapter 6 discusses use of the DATA statement to establish initial values for arrays, array elements, substrings, and variables.
- Chapter 7 describes assignment statements and their use to establish values for variables, substrings, and array elements.
- Chapter 8 describes the functions and operations of control statements.
- Chapter 9 discusses files, units, input/output (I/O) statement components, data transfer I/O statements, and auxiliary I/O statements.
- Chapter 10 defines format specification descriptors and their application for defining the format of data being read or written.
- Chapter 11 discusses both intrinsic and user-written subprograms, including block data subprograms and procedure subprograms (functions and subroutines).
- Chapter 12 discusses the SPP Series synchronization features CONVEX Fortran supports with library routines, compiler directives, and intrinsic routines.
- Chapter 13 discusses the various optimization options available for use with the SPP Series Fortran compiler.
- Appendix A lists CONVEX Fortran's generic and specific intrinsic functions in table form and lists commonly used library routines.
- Appendix B discusses Fortran 66 compatibility.
- Appendix C discusses Fortran 90 compatibility.
- Appendix D discusses Cray Fortran compatibility.
- Appendix E discusses VAX Fortran compatibility.
- Appendix F discusses HP Fortran compatibility.
- Appendix G discusses Sun Fortran compatibility.

An index is included at the end of this manual.

Tables

How to use this reference

Purpose and audience

This publication is a reference for the CONVEX Fortran programming language and is designed to provide a thorough working definition of the language. This manual presents information about the CONVEX Fortran compilers available for both the CONVEX C Series and the CONVEX SPP Series architectures. Some information is specific to one hardware platform and is marked to indicate this within the text.

CONVEX Fortran is an implementation of the ANSI FORTRAN 77 standard that includes many extensions and enhancements not present in the standard. This manual encompasses both the *American National Standard programming language Fortran* (ANSI X3.9-1978) document and the CONVEX extensions to FORTRAN 77.

It is assumed throughout that you are an experienced Fortran programmer. For further discussion of the CONVEX Fortran language and other CONVEX software, refer to the "Associated documents" section in this preface.

Although a detailed knowledge of the operating system on the CONVEX computer you use is not necessary to understand this document, some familiarity with the system is beneficial. If you are unfamiliar with the operating system, consult the "Associated documents" section in this chapter.

Organization

This reference is organized as follows:

- Chapter 1 discusses Fortran program elements and program unit format.
- Chapter 2 discusses data types, constants and variables.
- Chapter 3 describes arrays and substrings.
- Chapter 4 discusses expressions and how they are used to compute and evaluate values.

- Chapter 5 discusses specification statements.
- Chapter 6 discusses use of the DATA statement to establish initial values for arrays, array elements, substrings, and variables.
- Chapter 7 describes assignment statements and their use to establish values for variables, substrings, and array elements.
- Chapter 8 describes the functions and operations of control statements.
- Chapter 9 discusses files, units, input/output (I/O) statement components, data transfer I/O statements, and auxiliary I/O statements.
- Chapter 10 defines format specification descriptors and their application for defining the format of data being read or written.
- Chapter 11 discusses both intrinsic and user-written subprograms, including block data subprograms and procedure subprograms (functions and subroutines).
- Chapter 12 discusses the SPP Series synchronization features CONVEX Fortran supports with library routines, compiler directives, and intrinsic routines.
- Chapter 13 discusses the various optimization options available for use with the SPP Series Fortran compiler.
- Appendix A lists CONVEX Fortran's generic and specific intrinsic functions in table form and lists commonly used library routines.
- Appendix B discusses Fortran 66 compatibility.
- Appendix C discusses Fortran 90 compatibility.
- Appendix D discusses Cray Fortran compatibility.
- Appendix E discusses VAX Fortran compatibility.
- Appendix F discusses HP Fortran compatibility.
- Appendix G discusses Sun Fortran compatibility.

An index is included at the end of this manual.

Scope

This reference covers CONVEX Fortran Version 9.1, which runs on both the CONVEX C Series hardware platforms and the CONVEX SPP Series platform.

The CONVEX Fortran compiler runs under ConvexOS Version 11.0 or higher (on C Series machines) and under SPP-UX Version 2.1 or higher (on SPP Series machines).

Notational conventions

The following conventions are used throughout this document:

- Brackets ([ ]) designate optional entries.
- A caret (^) represents the space character.
- Horizontal ellipsis (. . .) shows repetition of the preceding item(s). In an example, horizontal ellipsis indicates that statements are omitted.
- Vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- References to the online man pages appear in the form fc(1F), where the name of the manual page is followed by its section number enclosed in parentheses. See the "Online man pages" section of this preface for more information.
- *Italics* within text denote user-supplied variable information.
- `Monospaced text, like this`, is used to denote screen output, code examples, non-variable text in command forms, file names, utility names, and, in general, text that appears exactly as shown on output or must be entered exactly as shown on input.
- Within command sequences set apart from text, *italics* indicate user-supplied variable information. Substitute actual information for the italicized words. 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, or libraries.
- *Text describing CONVEX extensions to the Fortran language appear in this type style.* This convention is not followed in appendices that specifically deal with non-standard Fortran compatibility modes.

- Command sequences or forms which are CONVEX extensions appear in *monospaced italics, like this text*.

C Series only

This C Series only graphic identifies this paragraph, and this paragraph only, as C Series specific (including all C Series platforms).

SPP Series only

This graphic marks this paragraph, and only this paragraph, as specific to SPP Series (also known as Exemplar) machines.

Note

A note highlights information of a supplemental nature. The note immediately precedes or follows the highlighted information.

Caution

A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Associated documents

This section lists additional information resources recommended to both C Series and SPP Series Fortran programmers.

C Series and SPP Series publications

The publications that follow address both C Series and SPP Series programming issues.

- *Fortran User's Guide* (DSW-038) describes how to compile, run, debug, and analyze CONVEX Fortran programs.
- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251) (optional product) describes how to use the interactive profiler.
- *CONVEX CXdb Concepts* (DSW-471), *CONVEX CXdb User's Guide* (DSW-473) and the *CONVEX CXdb Reference* (DSW-472) describe all aspects of the optional CXdb debugger, which is briefly described in Chapter 4 of the *CONVEX Fortran User's Guide*.

C Series publications

C Series only

The following publications are recommended to CONVEX C Series programmers. The information in these references is specific to CONVEX C Series computers.

- *Fortran Optimization Guide* (DSW-034) describes the types of optimization available in CONVEX Fortran and shows you how to use optimization directives and options.

- *CONVEX Interlanguage Programming Guide* (DSW-043) outlines techniques for calling CONVEX Fortran routines from programs written in other languages, and for calling routines written in other languages from CONVEX Fortran.
- *ConvexOS Primer* (DSW-133) has basic self-instruction for learning and using the ConvexOS operating system.
- *CONVEX adb Debugger User's Guide* (DSW-009), a tutorial and reference manual, describes the functions and operations of the CONVEX `adb` debugger.
- *CONVEX Consultant User's Guide* (DSW-025) (optional product) describes the functions and operations of the CONVEX `csd` debugger, postmortem dump (`pmd`) utility, and the `gprof` profiler.
- *CONVEX Compiler Utilities User's Guide* (DSW-096) describes the CONVEX loader and the CONVEX assembler.
- *CONVEX COVUEshell Reference Manual* (DSW-136) (optional product) describes COVUEshell. COVUEshell is an optional CONVEX product that provides a VMS-type interface, giving the user access to a subset of Digital Command Language (DCL) commands.
- *CONVEX Application Compiler User's Guide* (DSW-401) (optional product) describes how to use the CONVEX Application Compiler to optimize programs.
- *CXmetrics User's Guide* (DSW-475) (optional product) describes software metrics data and explains how to use CONVEX CXmetrics to report on this data.

SPP Series publications

SPP Series only

The *Exemplar Programming Guide* is recommended to CONVEX SPP Series programmers. Information in it is specific to CONVEX SPP Series computers.

- *Exemplar Programming Guide* (DSW-067) describes efficient programming techniques for the Exemplar family of computers.

Other documents

Other documents of interest to all Fortran programmers include:

- *American National Standard programming language Fortran,* ANSI X3.9-1978. This book is the definition of standard

FORTRAN 77, which is fully supported in this release of CONVEX Fortran.

- ISO/IEC 1539:1991, the International Fortran Standard. This book is the definition of standard International Fortran, which is identical to the ANSI Fortran 90 programming language, ANSI X3.198-1992, certain features of which are supported in this release of CONVEX Fortran.

Online man pages

In addition to the references listed, the online man pages provide information useful to Fortran programmers. The `man` program formats and displays the information contained in the man pages.

Sections 1 and 7 of the man pages primarily contain operating system information for users. Sections 2 through 5 contain information for programmers. The following list summarizes the topics addressed in sections of the man pages:

- Section 1 — Commands of general utility and commands for communicating with other systems.
- Section 2 — System calls and error numbers.
- Section 3 — Various library functions.
- Section 4 — Special files, related driver functions, and networking support.
- Section 5 — Various file formats.
- Section 7 — Miscellaneous commands, primarily related to text processing and terminal environments.

References to the online man pages appear in the form fc(1F), where the name of the manual page is followed by its section number enclosed in parentheses.

You can access the online man pages by entering:

```
% man entryname
```

where *entryname* is the name of the man page to be displayed. For instance, entering `man fc` on the command line brings up the man page for `fc`, the Fortran compiler. For more detailed information, refer to the man(1) man page.

Technical assistance

If you have questions that are not answered by the documentation, contact the CONVEX Technical Assistance Center (TAC). To contact the TAC, use one of the following phone numbers:

- Within the continental U.S., call (800) 952-0379.
- Outside the continental U.S., contact the local CONVEX office.

The `contact` utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

Refer to the contact(1) man page for complete details.

Introduction 1

CONVEX Fortran is a high-level programming language that maximizes software portability and enhances the speed of execution using global and local optimization, vectorization (on CONVEX C Series machines), and parallelization techniques.

CONVEX Fortran includes standard Fortran features as defined by the American National Standard FORTRAN 77 (ANSI X3.9-1978) and unique CONVEX extensions. CONVEX Fortran also supports certain features of Fortran 90 and provides compatibility with selected features of the Cray, DEC, Hewlett-Packard, and Sun Fortran compilers.

This manual presents information about the CONVEX Fortran language available for both the CONVEX C Series and the CONVEX SPP Series architectures. Some information is specific to one hardware platform and is marked to indicate this within the text. *CONVEX extensions to the language are set in this style of type throughout this manual.*

The remainder of this chapter provides an overview of Fortran programs, their component parts, the Fortran character set, symbolic names permitted, and related information.

Types of programs

A Fortran program consists of a main program and, optionally, one or more subprograms. The main program is the basic program unit. Subprograms are separate program units that are called by the main program or other subprograms.

A main program begins with a `PROGRAM` statement. Subprograms, which include both "block data" subprograms and procedures (functions and subroutines), begin with either a `BLOCK DATA`, `FUNCTION`, or `SUBROUTINE` statement.

A program unit, such as a subprogram or main program, is defined as a sequence of Fortran statements that ends with an `END` statement. A program unit also can include comment lines

and may also include an `OPTIONS` *statement that establishes compiler options.*

Fortran statements

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

If you are entering your CONVEX Fortran program from a terminal, you can enter statement lines of any length as long as you do not exceed the compiler's statement length limit (refer to the *CONVEX Fortran User's Guide*, Appendix G, for the statement length limit) or use the *-72* compiler option, which instructs the compiler to evaluate only the first 72 characters of each statement.

Comment lines

Not every line in a Fortran program is a statement; some lines are comment lines. A comment line has no effect on the actual execution of a program. It is used for documenting program action, identifying processes, or improving program readability. You can place a comment line anywhere in a program unit, even before the initial statement 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 any column 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.

Executable statements

As mentioned earlier, executable statements specify action within a program and form a sequence of execution. Executable statements include the following:

- Arithmetic, logical, statement label (`ASSIGN`), and character assignment
- Unconditional `GOTO`, assigned `GOTO`, and computed `GOTO`
- Arithmetic `IF` and logical `IF`

- Block IF ELSE IF, ELSE, and END IF
- CONTINUE
- STOP and PAUSE
- DO and *ENDDO*
- WHERE, ELSEWHERE, and ENDWHERE
- *ALLOCATE* and *DEALLOCATE*
- READ, WRITE, *ACCEPT*, TYPE, and PRINT
- REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE
- CALL and RETURN
- END

Nonexecutable statements

Nonexecutable statements do not specify action. Instead, they define properties of variables; contain editing information; classify program units; and designate subprogram entry points. Nonexecutable statements include the following:

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

Program format

Each Fortran statement is divided into fields providing for statement label, continuation indicator, statement text, and a

sequence number. Table 1gives general rules for entering items into fields.

Table 1 Fortran fields

Column	Purpose	Use
1	Comment or debugging statement indicator	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 has 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 followed by a digit 1 through 9 indicate continuation.*
7-end of line	Fortran statements	This field contains the actual Fortran statement. *At any point in the field, except within character or Hollerith constants, an exclamation point indicates that the remaining text is a comment.*

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

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.

Statement label field

A statement label references statements in a program unit. Although any statement can have a label, only labeled executable statements and FORMAT statements can be referenced 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 through 5. Blanks and leading zeros are ignored; for example, `5`, `05`, and `00005` are the same label.

Initial line

An initial line begins a single Fortran statement. A 0 or a blank (space) in column 6 indicates an initial line. An initial line cannot be a comment line; however, it can have a statement label. If you do not label the line, leave columns 1 through 5 blank.

A Fortran statement may be comprised of a single initial line or an initial line and all following continuation lines (up to the next initial line). Comments and/or blank lines are allowable in between initial lines and continuation lines, among continuation lines, and in between statements.

Continuation line

A continuation line is a line with no statement label that contains blanks in columns 1-5, and any character other than a blank or 0 in column 6. A continuation line can also begin with a single tab character followed by a digit 1-9.

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, except within character or Hollerith constants.* (Refer to the *CONVEX Fortran User's Guide*, Appendix F, 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 (;), can 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.

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. Refer to the CONVEX Fortran User's Guide, Appendix A, for more information.*

Debug statements

With CONVEX Fortran you can place a debugging statement indicator, the letter `D`*, in column 1 of the statement label field. You can 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 can 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.*

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 to be the first character of the statement field, as if it had been entered in column 7 in character-per-column format.

ANSI-standard formatting

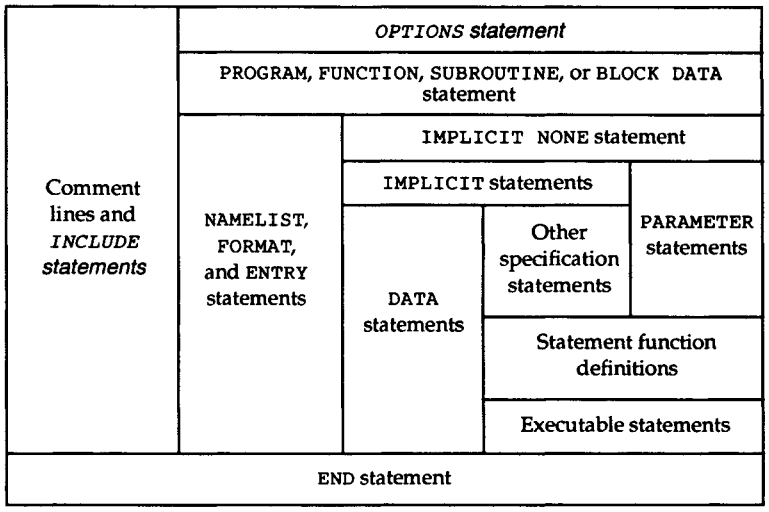
The ANSI Fortran standard specifies a restricted form of character-per-column formatting and states that all lines are 72 characters in length. *You can achieve the same effect with CONVEX Fortran by specifying the* `-72` *option on the compiler command line. If* `-72` *is specified, any line shorter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

Order of statements and lines

Figure 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 Required order of statements



Fortran character set

The standard FORTRAN 77 character set consists of the following characters:

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

The CONVEX extended character set includes the following:

Lowercase letters	a through z
Exclamation mark	!
Percent sign	%
Ampersand	@
Quotation mark	"
Underscore	_
Left angle bracket	<
Right angle bracket	>

Pound sign	#
Semicolon	;
Tab	

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

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

Symbolic names

Variables, arrays, and functions have symbolic names that identify them in a program. A symbolic name must start with a letter and can be followed by any number of uppercase letters (A-Z), digits (0-9), *lowercase letters (a-z), underscores (_), or dollar signs ($), up to the maximum length allowed for a symbolic name (refer to the CONVEX Fortran User's Guide, Appendix F, for the maximum name length).*

The compiler converts letters specified in lowercase (a-z) to uppercase; thus the symbolic names "ABC" and "abc" are the same. Because dollar signs are used in CONVEX-supplied software, CONVEX recommends that you not use dollar signs in symbolic names.

Data types, variables, and constants

2

The symbolic name associated with each variable, constant, array, and function contained in a Fortran program is assigned a data type, either implicitly or explicitly. This chapter discusses CONVEX Fortran data types and their application to variables and constants.

Data types

The *American National Standard programming language Fortran* defines five data types: CHARACTER, COMPLEX, INTEGER, LOGICAL, and REAL. Table 2 shows the ANSI Fortran data types available in CONVEX Fortran, along with their CONVEX equivalents.

Table 2 Data types

ANSI data type	Corresponding CONVEX data types
CHARACTER	CHARACTER*n, CHARACTER*(*)
COMPLEX	COMPLEX*8, *COMPLEX*16*, *DOUBLE COMPLEX*
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, *REAL*16*†

†*REAL*16* is supported on CONVEX SPP Series machines and in native mode on CONVEX C Series machines.

CONVEX Fortran also supplies the GATE *and the* BARRIER *data types for use with certain instrinsics and compiler directives on*

CONVEX SPP Series machines. For information about gates and barriers, refer to Chapter 12, "SPP Series synchronization."

CONVEX Fortran supports the `RECORD` *derived data type as an extension. Use of the* `RECORD` *type is discussed in detail in Appendix E, "VAX Fortran compatibility."*

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* `DOUBLE COMPLEX` *(or* `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`.

For `LOGICAL` *and* `INTEGER` *data types, the storage requirement can be controlled by the* `-p8` *or* `-pd8` *compiler options; the default is four bytes. For* `REAL` *and* `COMPLEX`*, the storage requirements can be controlled by the* `-p8` *and* `-pd8` *compiler options.*

For the `CHARACTER` data type, *n* can have an integer value ranging from 1 to the maximum length permitted by the system (refer to Appendix G, "System limits," of the *CONVEX Fortran User's Guide*). The notation `CHARACTER*(*)` specifies an assumed-length character string.

Table 3 shows the storage requirements for each data type. Defaults appear inside parentheses, but the defaults change when you use the `-p8`, `-pd8`, or `-cfc` compiler options. Refer to Chapter 1 of the *CONVEX Fortran User's Guide* for more information on compiler options.

Some data types can be assigned either explicitly or implicitly. Type-declaration statements are used for explicit typing; refer to the "Type-declaration statements" section of Chapter 5, "Specification statements," for details. Symbolic names that are not explicitly typed are typed implicitly according to their first letter; names beginning with the letters I through N are of the default `INTEGER` type; all others are of the default `REAL` type. The `IMPLICIT` statement allows you to override these implicit type assignments; refer to the "`IMPLICIT` statement" section of Chapter 5, "Specification statements," for more information.

Table 3 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), *8, or 16*†
REAL*4	4
REAL*8	8
*REAL*16*	*16*†
COMPLEX	(8) *or 16*
COMPLEX*8	8
*COMPLEX*16*	*16*
DOUBLE PRECISION	(8) *or 16*†
DOUBLE COMPLEX	(16)
CHARACTER	1
CHARACTER**len*	*len*
CHARACTER*(*)	
RECORD	*Varies*

†The REAL*16 data type (including DOUBLE PRECISION*16) is available on CONVEX SPP Series machines and in CONVEX native mode on C Series.

Conversion of data types

Where data types differ 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. The compiler uses the following rules for conversion:

- *Treats LOGICALS as INTEGERS of the same length.*
- *Converts INTEGERS to longer types by sign extension and to shorter types 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 a REAL value to a lower precision REAL value (for example, REAL*8 to REAL*4) by rounding to the lower precision.
- Converts a REAL value to a higher precision REAL value by zero-extending the mantissa. Converting a REAL value to a higher precision, however, does not increase the accuracy of the value.
- 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.*

On C Series machines, optimizing code that contains type conversions (including implicit type conversions) can cause some code to vectorize poorly or not at all.

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 (refer

to Chapter 6, "DATA statement") or a type-declaration statement (refer to Chapter 5, "Specification statements"). A variable is associated with a storage location. Whenever a variable is used, the current value in the storage location is referenced.

Variable types are classified and assigned according to the methods discussed in the "Data types" section of this chapter.

Multiple variables can be associated with the same storage location by using 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 (refer to Chapter 5, "Specification statements").

Constants

A constant is a scalar arithmetic or logical value or a character string. It does not change during program execution. The form in which a constant is expressed determines the value and data type. The PARAMETER statement assigns a symbolic name to a constant. Refer to Chapter 5, "Specification statements," for more information on the PARAMETER statement.

INTEGER constants

An INTEGER constant consists of the digits 0 through 9 and, possibly, a leading sign. The sign is optional for a positive constant but required for a negative constant. Leading zeros have no effect on the value.

Examples:

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

INTEGER constants take the default INTEGER precision. If the default precision is 2 or 4 bytes and the constant is too large, it is treated as an 8-byte constant; if it is too large to be represented in 8 bytes, the compiler truncates the constant to fit and issues a warning. The -cfc, -p8, and -pd8 compiler options affect the

default precision. Refer to Chapter 1 of the CONVEX Fortran User's Guide for more information.

REAL constants

A REAL 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 (for example, 2.1, .2, 678912.). When you specify an exponent, the decimal is optional; `7.E3` is the same as `7E3`.

*A DOUBLE PRECISION constant is similar to REAL except that an exponent is required and the exponent letter D is used. REAL*16 data requires the exponent letter Q. If the Q is omitted, the value defaults to REAL.*

REAL and DOUBLE PRECISION constants take the default precision for their corresponding variable types. If the constant is too large, the compiler issues a fatal error. The -cfc, -p8, and -pd8 compiler options affect the default precision. Refer to Chapter 1 of the CONVEX Fortran User's Guide for more information.

Examples:

Valid	Invalid	Reason
`2500.`	`2500`	Decimal point missing
`+2.0E2`	`2.0E`	Exponent field missing
`5E4`	`5,000`	Has comma
`4E-2`		
`3.0E4`		
`5.4Q4`		

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*2, INTEGER*4, or REAL*4 constants. A *COMPLEX*16*

constant is an ordered pair of integer, `REAL*4` or `REAL*8` constants.

Example:

Valid	*Invalid*	*Reason*
`(1.6405D0,-1.6405D0)`	`(1.640D)`	*Second constant missing*

`COMPLEX` constants take the default `COMPLEX` precision. If the constant is too large, the compiler issues a fatal error. The `-cfc`, `-p8`, and `-pd8` compiler options affect the default precision. Refer to Chapter 1 of the CONVEX Fortran User's Guide for more information.

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 can be in the following forms

*'cc...c'*O	*standard octal notation*
O*'cc...c'*	*standard octal notation*
"cc...c	*VAX octal notation*
cc...c*B*	*Cray Boolean octal notation*

where

c represents an octal digit. For more information about Cray or VAX Fortran compatibility, refer to Appendix D, "Cray Fortran compatibility," or Appendix E, "VAX Fortran compatibility."

Examples:

Valid	*Invalid*	*Reason*
`'765'O`	`'835'O`	*8 not in range 0 to 7*
`'123'O`	`123O`	*Missing apostrophes*

Hexadecimal constants

A hexadecimal constant consists of one or more hexadecimal digits enclosed in apostrophes and followed or preceded 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
```

or

```
X'cc...c'
```

or

```
Z'cc...c'
```

where c represents a hexadecimal digit.

Examples:

Valid	*Invalid*	*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*
`X'66F'`	`X129A'`	*Missing first apostrophe*
`'abc123ff'X`		

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

- *When octal or hexadecimal constants are used 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.

Example:

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

Truncation/extension occurs as shown below:

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

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
```

or

```
nLcc...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 at least one character*

The Cray form (`nLcc...c`) left-justifies the constant in memory and zero-fills its storage space to the right. Refer to Appendix D, "Cray Fortran compatibility," for more information.

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 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.*
- *If you pass a Hollerith constant to a subroutine, you must pass it into a dummy argument of type `INTEGER`, `REAL`, or `LOGICAL`. Passing it into a `CHARACTER` variable will cause erroneous behavior. Refer to the ANSI FORTRAN 77 standard, page C-3, for more information.*
- *In any other context, the constant assumes an `INTEGER*4` data type, unless you change the default `INTEGER` length by using the `-p8`, `-pd8` or `-cfc` options.*
- *When a Hollerith constant continues across a line, you can use the -72 option to imitate the behavior of other Fortran compilers (blank padding to column 72).*

If you continue a Hollerith constant on another line, you must use the `-72` option for compilation.

`LOGICAL` constants

A `LOGICAL` constant represents the value true or false. A value of true is represented by `.TRUE.` and a `LOGICAL` constant assigned to `.FALSE.` represents value of false.

CHARACTER constants

A CHARACTER constant is a string of printable ASCII characters enclosed within 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 one apostrophe.

Examples:

```
'final'
'two''s complement'
```

You can delimit a CHARACTER constant with 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 one quotation mark.

Examples:

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

If you continue a CHARACTER constant on a second line, you must use the -72 option for compilation.

Constant expressions using intrinsics

CONVEX Fortran supports the use of the following generic intrinsics (and their associated specific intrinsics) in constant expressions:

IAND	IOR	NOT
IEOR	ISHFT	LGE
LGT	LLE	LLT
MIN	MAX	ABS
MOD	ICHAR	NINT
DIM	DPROD	CMPLX
CONJ	IMAG	

When the compiler evaluates a constant expression that has an intrinsic, it determines the constant's data type based on the intrinsic's result type. For example, the following code declares the constant LOW by using the AMIN1 intrinsic routine (AMIN1 is a specific form of the generic intrinsic MIN):

```
PARAMETER (PR1=32.59, PR2=6.82, PR3=12.65)
PARAMETER (LOW = AMIN1(PR1, PR2, PR3))
```

These statements assign LOW the REAL*4 value 6.82. The first expression establishes values for three REAL constants, PR1, PR2, and PR3. The second expression assigns LOW the result 6.82, returned by the AMIN1 intrinsic. Because AMIN1 returns a REAL*4 result, LOW takes the type REAL*4.

For more information about CONVEX intrinsic routines and their result types, refer to Appendix A, "Intrinsics and commonly used library routines."

Arrays and substrings 3

An array is a set of storage locations of the same data type identified by a single name and accessed individually by a subscript. A substring is a contiguous portion of a character string.

This chapter discusses both arrays and substrings in detail. More information about CONVEX Fortran support for Fortran 90 array conventions, including information about methods of manipulating arrays and substrings, is available in Appendix C, "Fortran 90 compatibility."

For information about array-valued functions consult Chapter 11, "Subprograms." Information about declaring arrays of type *GATE* or *BARRIER* appears in Chapter 12, "SPP Series synchronization."

Arrays

Conceptually, a one-dimensional array is a column of elements where each element is accessible by its subscript. A two-dimensional array can be thought of as a table of elements containing rows and columns, with each element accessible by a pair of subscripts. *You can reference an entire array by using the array name with no subscripts, and you can reference sections of an array using subscript ranges.*

An array can have from one to seven dimensions, with the number of subscripts used to reference individual elements equal to the number of dimensions. Refer to the "Referencing array elements" section of this chapter for more information.

All the values in an array are of the same data type and any value assigned is converted to the array's data type. 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 statement or an input statement. Likewise, an entire array can be defined *with an assignment statement* or with an input statement.

Array rank and shape

The number of dimensions of an array is referred to as the array's rank. An array can be described as being rank n, where n is the number of dimensions. Related to this is the array's shape, which describes the absolute number of elements for each dimension. Shape has the form $(m_1, m_2, \ldots m_n)$, where m is the number of elements in the given dimension, and n is the rank of the array.

Arrays having the same shape are said to be conformable. Scalar values are conformable with any shape array; when a scalar is used in this context, it can be thought of as occupying an array of the proper shape with its value in each element of the array.

Rank, shape, and conformability are important in discussing Fortran 90 array support, which CONVEX Fortran provides in a limited capacity. Supported Fortran 90 array features are discussed under applicable topics in this chapter; all Fortran 90 features present in CONVEX Fortran are discussed in detail in Appendix C, "Fortran 90 compatibility."

Types of arrays

CONVEX Fortran permits three types of arrays: static-sized arrays, *allocatable arrays*, and *automatic arrays*. Both *allocatable* and *automatic arrays* are provided as part of CONVEX Fortran's Fortran 90 support. All three types of arrays are briefly described here. More detailed information is available in the remainder of this chapter and in Appendix C, "Fortran 90 compatibility".

Static-sized arrays

The rank and shape (size) of static-sized arrays is defined at compile time and does not change during a program's execution. Modifying the rank, shape, or size of a static-sized array involves modifying and re-compiling the source code in which it is defined.

Unlike the other array types, static-sized arrays may appear in COMMON blocks and SAVE statements. On CONVEX C Series machines static-sized arrays are saved by default. On SPP Series machines Fortran does not save static-sized arrays by default.

Allocatable arrays

An allocatable array can change shape during program execution and is local to the subroutine in which it is declared. Although an allocatable array keeps the same rank (number of dimensions) throughout the program, storage for it is

dynamically allocated on the heap at runtime, so the size of the array's dimensions can change.

When storage allocated to an allocatable array is deallocated (released), the array can be reallocated to occupy an amount of storage which may differ from the previous allocation.

Automatic arrays
An automatic array is local to a subroutine and contains one or more nonconstant dimensions. Memory for an automatic array is allocated on the heap on entry into the subroutine; this storage is released on exit from the subroutine. Automatic arrays cannot be saved using the SAVE *statement, so all information stored in an automatic array is lost on exit from the subroutine in which it is used.*

Array storage

Even though array elements are arranged and referenced in dimensions, array storage in memory is linear. For example, a one-dimensional array is a column of figures, stored with the first element in the first storage location and the last element in the last storage location of the sequence.

Multidimensional array elements are stored so that the value of the first (leftmost) subscript varies most rapidly. For example, `A(1,1)`, `A(2,1)`, and `A(3,1)` are stored in contiguous memory locations. This method of storage is called "column-major order." In some other languages, such as C and Ada, arrays are stored in row-major order, where the rightmost subscript varies most rapidly.

Character substrings

A character substring is a sequence of contiguous 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 following form:

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

where *var* is a CHARACTER variable or array name, *subscript* is an array subscript (required only when *var* is an array), *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. Character positions are numbered from left to right within a CHARACTER variable or array element.

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 *expr2*. The value of *expr2* must be less than or equal to the length of the string.

Given the following statements:

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

the reference:

```
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).

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

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

Declaring arrays

An array declarator defines the name of the array within the program unit, the number of dimensions (the array's rank), and the upper and lower bounds of elements in each dimension (its shape). For multidimensional arrays, separate the dimension declarators with commas. DIMENSION, COMMON, *ALLOCATABLE*, *POINTER*, and type statements allow array declarations.

For information about declaring arrays of type *GATE* or *BARRIER*, refer to Chapter 12, "SPP Series synchronization."

CONVEX Fortran supports the use of automatic arrays. Any array declared local to a subroutine with one or more non-constant dimensions is considered an automatic array. Automatic arrays are discussed in this section and in the "Automatic arrays" section later in this chapter.

Array declarations have the following form:

type arrname([*lb*:] *ub* [, ...])

or

DIMENSION *arrname*([*lb*:] *ub* [, ...]) [, *arrname*...]

where

type

is the type of the array elements. Any valid CONVEX Fortran type can be used. If the DIMENSION form is used, the array will take an implicit type based on its name if it is not explicitly typed elsewhere.

arrname

is the name of the array. Multiple arrays can be dimensioned in a single DIMENSION statement. *If the array is to be automatic, arrname must be local to a subroutine.*

lb

is the lower dimension bound. The default is 1.

For static arrays, *lb* must be a constant. *For allocatable and automatic arrays, lb can be a constant, variable or expression.*

ub

is the upper dimension bound. *ub* must be greater than or equal to *lb*.

For static arrays, *ub* must be a constant. *For allocatable and automatic arrays, ub can be a constant, variable or expression.*

If you do not specify a lower bound, the default value is 1, and the upper bound is the number of elements given for the 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 with a colon.

For automatic arrays, ub and/or lb for at least one dimension must either consist of or be derived from an integer argument passed to the subroutine in which the automatic array resides.

Different dimensions of a multidimensional automatic array can be declared to be different sizes based on one or more arguments passed to the subroutine.

The type of array and the product of the subscripts determine the number of storage units allocated to each array named in the statement. Fortran arrays can have from one to seven dimensions.

Static arrays

A static array cannot change rank or shape during runtime. The examples in this section illustrate several forms of static array

declarations. For more information about declaring static arrays, refer to the "Declaring arrays" section of this chapter.

```
DIMENSION MYRAY(5)
```

This example dimensions a one-dimensional static array with five elements. The array is implicitly typed INTEGER.

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

The above example dimensions a one-dimensional static INTEGER array with five elements numbered 0 through 4.

```
COMMON RERAY(5,5)
```

This example allocates storage in an unnamed COMMON block for a two-dimensional static array with 25 elements (5 rows by 5 columns). The array is implicitly typed REAL.

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

This example specifies a two-dimensional static INTEGER array with 25 elements. Elements in the first dimension are numbered -1 through 3; elements in the second are numbered 2 through 6.

```
INTEGER A(5,5,5)
```

This example declares a three-dimensional static array with 125 elements (5 planes, 5 rows, and 5 columns) of type INTEGER. The INTEGER type statement overrides implicit typing.

```
CHARACTER*8 MRAY(2)
```

This example declares a one-dimensional static CHARACTER array with two elements of eight bytes each.

Allocatable arrays

CONVEX Fortran supports the use of Fortran 90 allocatable arrays. Storage for allocatable arrays is dynamically allocated on the heap at runtime when an ALLOCATE statement is executed. The heap space must be deallocated through use of the DEALLOCATE statement when the allocatable array is no longer needed. Allocatable arrays must be declared as such with the

other specification statements in the program before they can be allocated.

The ALLOCATABLE statement

Allocatable arrays must be declared with the ALLOCATABLE *statement. The array's rank must be supplied either in the* ALLOCATABLE *statement, in a* DIMENSION *statement, or in the array's type declaration, which, if used, precedes the* ALLOCATABLE *statement. Rank definitions have the following form:*

```
arrname ( : [ , : . . . ] )
```

where arrname *is the name of the array, which is followed by parentheses containing one colon for each dimension of the array. Multiple colons are separated by commas. Code examples of this are given in Appendix C, "Fortran 90 compatibility."*

Note that the above example is not an executable statement, but rather a form for use within certain executable statements where a rank definition is required.

Allocatable array declarations have the following form:

```
[type arrexp ]
ALLOCATABLE  (arrexp  [ , . . . ] )
```

where

type

> *is an optional type definition for the array. The form for this is identical to the form for a standard array type definition, except instead of specifying constant dimension parameters, you must either supply a rank definition (number of dimensions) or provide no parameters.*

arrexp

> *is the array name or rank definition if it was not given in a preceding type statement. The array name may occur in both the type and* ALLOCATABLE *statements; the rank definition must occur in exactly one, or the compiler reports an error.*

Refer to Appendix C, "Fortran 90 compatibility," for a detailed example of an allocatable array declaration.

Example:

```
INTEGER A(:), B
ALLOCATABLE (A, B(:,:), X(:,:,:),I(:))
```

In this example, arrays A *and* B *are declared type* INTEGER *and* A *is given rank one in the type declaration. In the* ALLOCATABLE *statement, then,* A *cannot be given a rank.* B *is given rank two, and it will take the type* INTEGER *from the type statement above.* X *is given rank three and implicitly typed* REAL. I *is implicitly typed* INTEGER, *and given rank one.*

The ALLOCATE and DEALLOCATE statements

After declaring an allocatable array, you must allocate storage for it before you can use it. This is done with the ALLOCATE *statement, which has the following form:*

```
ALLOCATE (arrdef [, ...])
```

where arrdef is an array definition with dimension values or ranges for all dimensions. Dimension ranges are described under the "Declaring arrays" section of this chapter.

Example:

```
ALLOCATE (A(6),B(-9:0,0:9),X(5,10,-50:-40),I(10))
```

Recall that the array types have either been declared in preceding type statements or are implicitly typed. A *is allocated space for a 6-element one-dimensional array;* B *is allocated space for a 10-by-10 two-dimensional array, with row subscripts numbered from -9 to 0 and column subscripts numbered from 0 to 9;* X *is allocated space for a 5-by-10-by-11 three-dimensional array, with subscripts of the last dimension numbered from -50 to -40; and* I *is allocated space for a ten-element one-dimensional array.*

When you are done using an allocatable array, deallocate the array's storage space using the DEALLOCATE *command. You can deallocate multiple arrays with one statement, or, if you finish with them at different points in the program, individually. The* DEALLOCATE *statement has the following form:*

```
DEALLOCATE (arrname [, ...])
```

where arrname is the name of the array to be deallocated. arrname must be free of subscripts.

```
DEALLOCATE(A, B, X, I)
```

This example deallocates space for the arrays A, B, X, *and* I. *Because information contained in a deallocated array is lost, an error occurs if you try to access a deallocated array.*

You must manually deallocate arrays that are allocated local to a subroutine before exiting the subroutine.

To change the size or subscript range of a presently allocated allocatable array, you must first deallocate the array, then allocate it with the new information.

Automatic arrays

Any array declared local to a subroutine with one or more non-constant dimensions is considered an automatic array. Memory for automatic arrays is dynamically allocated on the stack on entry into the subroutine based on a variable dimension value passed into the subroutine (this is explained in detail further on). This stack space is freed on exit from the subroutine; automatic arrays cannot be saved using the SAVE *statement.*

Automatic arrays are declared with the following form:

type arrname(*n*[, . . .])

or

DIMENSION *arrname*(*n*[, . . .])

where

type

is any valid CONVEX Fortran type statement, such as INTEGER, REAL, COMPLEX, CHARACTER*n, *and so on.*

arrname

is the name of an array that is local to a subroutine.

n

is a constant, variable, or expression consisting of or derived from an integer argument passed to the subroutine in which the automatic array resides. n *represents the desired size of a certain dimension of the automatic array.* n *can be of the form* lb:ub, *where* lb *and* ub *specify dimension bounds as defined for conventional arrays.* lb *and* ub *can be constants, variables or expressions.* ub *must be greater than or equal to* lb. *At least one of the specified dimensions must be non-constant for* arrname *to be an automatic array.*

Different dimensions of a multidimensional automatic array can be declared to be different sizes based on one or more dimension arguments passed to the subroutine.

The following example illustrates two ways of declaring multidimensional automatic arrays.

```
SUBROUTINE SUB(I,J,K)
      .
      .
      .
!IMPLICITLY TYPED INTEGER
DIMENSION IARR(I,J,2*I)

!EXPLICITLY TYPED REAL
REAL*8 X(J+I,K)
      .
      .
      .
```

In this example the arrays `IARR` *and* `X` *are local to the subroutine* `SUB`. `I`, `J` *and* `K` *are integer arguments passed to the subroutine that are used in allocating the arrays. The arrays are automatically allocated at runtime on entry into the subroutine.* `IARR` *has one dimension of length* `I`*, one of length* `J`*, and one of length* `2*I`. `X` *has one dimension of length* `J+I` *and one of length* `K`*. Both are automatically deallocated on exit from the subroutine.*

Automatic arrays can only be used in subroutines and functions; they are not allowed in `MAIN` *programs or* `BLOCK DATA` *subprograms or in* `COMMON` *blocks, nor can they be used as dummy arguments.*

Referencing array elements

Individual array elements are referenced by subscripts. A pair of parentheses that encloses one to seven subscript expressions separated by commas constitutes a subscript. The subscript immediately follows the array name. Specify one subscript expression for each dimension defined for the array. For a two-dimensional array declared as `COMMON MYRAY(5,5)`, a valid reference is `MYRAY(2,4)`; an invalid reference is `MYRAY(5,7)`, because 7 lies outside the subscript range of the second dimension of the array. A subscript expression can be any valid arithmetic expression, *or a Fortran 90 array section.*

An array section is a piece of an array bounded by specified elements in each dimension. Array sections can be as small as

one element or as large as the entire array. An array section is specified with the following form:

```
arrname ( [i] : [j] [ :step] [ , ... ] )
```

where

arrname

is the array name.

i

is a constant, variable, or expression describing the element at which the section starts for that particular dimension of the array. i defaults to the declared lower bound of that dimension of the array.

j

is a constant, variable, or expression describing the element at which the section ends for that particular dimension of the array. j defaults to the declared upper bound of that dimension of the array.

step

is a constant, variable, or expression describing the number of elements to step along that particular dimension when selecting elements for the array section. The default step is 1.

Given the defaults for each of these values, array section specifiers such as `X( : )` *and* `X( : :1)` *are allowed. These particular examples are equivalent to the entire array, which can also be denoted by* `X`*.*

For a more detailed discussion of array sections and for examples of their use, see Appendix C, "Fortran 90 compatibility."

Expressions

4

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 either a scalar value *or an array value*. The operators determine which operations are executed on the operands. CONVEX Fortran supports four types of expressions: arithmetic, relational, LOGICAL, and CHARACTER.

Arithmetic expressions

An arithmetic expression includes arithmetic operators and arithmetic operands and produces a numeric value.

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, *array sections or* arithmetic function references. *The term numeric as used here includes* LOGICAL *data. The compiler treats* LOGICAL *data as* INTEGER *data when it is used in an arithmetic context.*

The arithmetic operator specifies the computation to perform on the operands. This computation generates a numeric value. Arithmetic operators are listed in Table 4.

Table 4 Arithmetic operators

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

Operator precedence

When an expression has two or more operators and contains no parentheses, the operations are executed in the order given in Table 5.

Table 5 Arithmetic operator precedence

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 has two or more operators of equal precedence, evaluation occurs in left-to-right order, except for exponentiation, which is evaluated right to left. CONVEX

Fortran, however, can 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 expressions, the innermost expression enclosed within parentheses is evaluated first.

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 type ordering is shown in Table 6.

Table 6 Data type priority

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

*When LOGICAL and INTEGER types 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, REAL*16, and COMPLEX*16. On CONVEX C Series machines the REAL*16 data type is available only in native mode.*

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.*

- *COMPLEX operations involving COMPLEX*8 and REAL*8 elements are evaluated as COMPLEX*16 operations; therefore, the REAL*8 element is not rounded.*

These conventions also apply to arithmetic operations where one of the operands is a constant.

Relational expressions

A relational expression compares the value of two arithmetic or two character expressions. Relational expressions produce a logical value of true or false.

In a relational expression, the two expressions being compared are separated by one of the relational operators shown in Table 7. Each relational operator must include delimiting periods.

Table 7 Relational operators

Relational operator	Comparison
.LT.	Less than
.LE.	Less than or equal
.EQ.	Equal
.NE.	Not equal
.GT.	Greater than
.GE.	Greater than or equal

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:

- LOGICAL or *INTEGER* constant
- LOGICAL or *INTEGER* variable
- LOGICAL or *INTEGER* array element
- LOGICAL or *INTEGER* expression enclosed in parentheses
- LOGICAL or *INTEGER* function reference
- Relational expression

The evaluation of a logical expression that has two or more LOGICAL operators is based on operator precedence as shown in Table 8.

Table 8 LOGICAL operator precedence

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

†Evaluation of the .OR. and .AND. operators is short-circuited. Refer to Chapter 8, "Control statements," for more information.

The LOGICAL operator .XOR. is the same as .NEQV. Operators on the same level of precedence are interpreted from left to right. Arithmetic rules for operator precedence apply for evaluation. Mixed operations are evaluated first by arithmetic rules of precedence, next by relational operations, and last by LOGICAL operations. *LOGICAL operands appearing in* IF *conditionals are short circuited; refer to the "Short circuit evaluation of conditionals" section of Chapter 8, "Control statements."*

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.*

CHARACTER expressions

The evaluation of a CHARACTER expression results in a string of type CHARACTER. Within a CHARACTER expression, two slashes can be used to specify concatenation. Concatenation produces a string that is a combination 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.

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

Specification statements 5

This chapter discusses specification statements, the nonexecutable statements that appear before the first executable statement in a program unit.

Specification statements define the type of a 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:

- PROGRAM
- COMMON
- IMPLICIT
- INCLUDE
- OPTIONS
- PARAMETER
- Type-declaration statements:
  - Numeric
  - Character
  - Record
- *POINTER*
- DIMENSION
- *ALLOCATABLE*
- EQUIVALENCE
- *STATIC*
- *AUTOMATIC*
- *NAMELIST*
- EXTERNAL

- INTRINSIC
- SAVE

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 6, "DATA statement."

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.

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 can 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 followed 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 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 the same COMMON block can be initialized in different program units.

CONVEX Fortran supports Cray TASK COMMON statements. Refer to Appendix D, "Cray Fortran compatibility," for more information.

COMMON block packing

Two methods are available for storing COMMON blocks: "loose" packing, which aligns all COMMON block data items to their natural boundaries, and "tight" packing, which permits COMMON data items to align to unnatural boundaries.

CONVEX Fortran supports loose packing for improved performance in certain situations and for compatibility with HP Fortran code. For information about CONVEX Fortran compatibility with HP Fortran see Appendix F of the Fortran Language Reference.

By default on CONVEX SPP Series machines, and when the `-align spp` compiler option is specified on C Series machines, the CONVEX Fortran compiler provides loose COMMON block packing by padding data items to their natural boundaries where necessary. This can improve performance in cases where partial-word items appear in COMMON blocks.

On C Series machines, or when the `-align cseries` compiler option is specified on SPP Series machines, COMMON blocks are stored using tight packing (the ANSI standard). Because this method does not pad COMMON block data items, partial-word items may cause other data items to align on non-8-byte boundaries. To optimize memory accesses in programs compiled using tight packing, data items appearing in COMMON blocks should be ordered from largest to smallest; quad-word (16 byte) items should appear first, followed by double-word items, single-word items, and finally partial-word items.

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]...)]...
```

or

```
IMPLICIT NONE
```

where

typ

is an INTEGER *[*len*]*, REAL *[*len*]*, DOUBLE PRECISION, *DOUBLE 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 (for example, 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 type REAL.

The `IMPLICIT NONE` form of the statement overrides all implicit defaults except intrinsic function types. When using `IMPLICIT NONE`, 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)
C     ANY NAME BEGINNING WITH THE LETTER F OR
C     ANY OF THE LETTERS H, I, J IS TYPE COMPLEX

      IMPLICIT LOGICAL(L)
C     ANY NAME BEGINNING WITH THE LETTER L IS TYPE
C     LOGICAL (VALUE  OF .TRUE. OR .FALSE.)

      IMPLICIT CHARACTER*8(C)
C     ANY NAME BEGINNING WITH C IS TYPE CHARACTER
C     WITH THE LENGTH OF THE CHARACTER ENTITY
C     BEING 8

      IMPLICIT REAL(A-H), (O-Z)
C     ANY NAME BEGINNING WITH THE LETTERS A
C     THROUGH H OR O THROUGH Z IS TYPE REAL
```

OPTIONS statement

You can use the OPTIONS statement to include options that are 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...]
```

A subset of the command line options can be specified through the OPTIONS statement. A list of these options appears in Fortran User's Guide Chapter 1.

```
OPTIONS -O2 -r8
```

This statement specifies that the following program unit is to be compiled at optimization level -O2 and that all REAL data that is not explicitly sized is to be compiled as DOUBLE PRECISION.

INCLUDE statement

The INCLUDE statement causes the compiler to insert source code from the specified file into the program being compiled. The contents of the file are inserted at the place where the INCLUDE statement appears. The statement has the form

```
INCLUDE 'filename'
```

or

```
#include "filename"
```

where filename is the path name of the file from which source code is to be read. The first form of the statement (without the # sign) is the recommended form; however, it cannot be used with the Fortran preprocessor (`fpp`).

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 F of the CONVEX Fortran User's Guide. 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.

PARAMETER statement

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

Standard PARAMETER statement

The PARAMETER statement assigns a symbolic name to a constant to be used within the program unit. It has the following form:

```
PARAMETER (name = exp[, name = exp] ...)
```

where:

name

is a symbolic name.

exp

is a constant or constant expression.

The *name* specified references 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 *except 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 before the PARAMETER statement. Constants of data type POINTER cannot appear in PARAMETER statements. If the length for a character-type constant is different from the default length, you must specify the length before the first appearance of the symbolic name.

Alternate PARAMETER statement

An alternate form of the PARAMETER statement also is available. CONVEX Fortran permits this alternate form to provide additional compatibility with VAX Fortran. For more information about VAX Fortran compatibility, refer to Appendix E, "VAX Fortran compatibility."

The 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. It 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, except as a real or imaginary part of a complex constant.

The symbolic name of a constant assumes the data type of its corresponding expression. Therefore you cannot specify the

data type of a parameter in a type-declaration statement, and the initial letter of the constant name does not affect the data type.

Example:

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

Type-declaration statements

Type statements are numeric, logical, CHARACTER, or *RECORD* type-declarations. Type statements override implicit typing and IMPLICIT statements. You can also use these statements to specify array dimensions.

Because variables of type GATE and BARRIER are declared using compiler directives rather than specification statements, they are not discussed here. For information about these data types, consult *Fortran Language Reference* Chapter 12, "SPP Series synchronization."

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 type-declaration statements.

Numeric type-declaration statements

Type-declaration statements have the form:

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

where

type

is any data type specifier except CHARACTER or *RECORD*.

v

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

clist

is a list of constants. (Refer to the "Data statement form" section of Chapter 6, "DATA statement.")

*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. This overrides the length attribute that the statement implies for the specified item. When you use data type-length specifiers with an array declarator, they can be placed immediately after data type or immediately after the array name, as in the following example:

```
REAL*8 X(100), Y(100)
```

or

```
REAL X*8(100), Y(100)
```

*Both of these declarations define X as a 100-element array of REAL*8 values; in the first, however, Y is also declared as a REAL*8 array. In the second, the data-type length specifier is specific to X, and provided the default REAL size hasn't been changed with an IMPLICIT statement or a compiler option, Y will be a REAL*4 (default REAL) array.*

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/
```

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 overrides the length specified in CHARACTER*_len_.

If you specify the length as an *, for example, CHARACTER*(*), a function name or dummy argument assumes the length specification of the corresponding function reference or actual argument, and a symbolic PARAMETER assumes the length of the actual constant. This is known as an assumed-length character argument.

RECORD type-declaration statements

CONVEX Fortran supports the RECORD data type as an extension under the -vfc compiler option. RECORD type declaration statements have the following form:

```
RECORD /structure-name/ record-namelist
```

where:

structure-name

is the name of a previously declared structure.

record-namelist

is a list of variable names, array names, or array declarations, separated by commas.

Structures and records are discussed in detail in the "Structure declaration" section of Appendix E, "VAX Fortran compatibility."

POINTER statement

The POINTER *statement allows you to declare one or more pointer variables, and to define their pointees. A pointer is a four-byte variable that contains the address of another variable or array, which is referred to as the pointee.*

The POINTER *statement has the following form:*

```
POINTER(p,s)
```

where

p

is the pointer being declared; it will hold the address of s. p *must be a variable, and cannot be declared as any other data type. Constants, arrays, statement functions and external functions cannot be used.*

s

is the pointee corresponding to p. s *contains the variable whose address is contained in* p. s *cannot be a pointer.* s *cannot be associated with any other known piece of named and referenced storage except through assignments to* p *or by associating two or more pointees with one pointer.* s *cannot be of type* CHARACTER.

Note

Associating two pointees with one pointer can inhibit optimization.

For purposes of arithmetic and data type conversions, the same rules that apply to variables of type INTEGER*4 *apply to pointers. Converting pointers to non-*INTEGER *variables should be avoided. Performing division and multiplication on pointers should also be avoided.*

A pointer can be passed into a subprogram as long as it is declared as a pointer in the subprogram.

The LOC function

The LOC *function can be used to assign the address of a variable into a pointer, as shown in the following code segment.*

```
POINTER(IPX, X)
IPX = LOC(Y)
```

In this example, X *and* Y *can be used to reference the same value. For more information on the* LOC *function, see the* LOC(3F) *man page.*

Dynamic memory allocation

Pointers provide a means by which CONVEX Fortran programs can call external routines that dynamically allocate memory. Other more automated dynamic memory allocation methods also are available; refer to Appendix C, "Fortran 90 compatibility," for more information.

The following example shows a CONVEX Fortran program calling a C routine that dynamically allocates an block of memory of a size specified by the user.

Fortran main program:

```
      PROGRAM DYNARRAY
      REAL A(1)          ! MUST DEFINE NUMBER OF
                         ! DIMENSIONS
      POINTER(IPA, A)    ! DECLARE POINTER
      INTEGER FALLOC     ! DECLARE C FUNCTION
      PRINT*, "Enter array size:"
      READ*, ISIZE       ! GET DESIRED ARRAY SIZE
      IPA = FALLOC(ISIZE*4)  ! ALLOCATE STORAGE
C ISIZE*4 BECAUSE DEFAULT REALS ARE 4 BYTES
      .
      .
      .
```

C routine:

```
int falloc_(int *size)/* trailing underscore
                      required for Fortran */
{
  int block;
  block = malloc(*size); /* allocate desired
                            memory w/malloc */
  return block;         /* return starting
                           address */
}
```

The array A is declared in DYNARRAY as a single-element, one-dimensional array because CONVEX Fortran needs to know the number of dimensions of an array, as well as the number of elements in each dimension except the last, at compile time. Since this is a one-dimensional array, the declared length of one element need not be static; it is superseded by the storage allocated when FALLOC is called.

For arrays declared with multiple dimensions, only the last dimension can be superseded by dynamically-allocated storage. However, dynamic multidimensional arrays can be created by dynamically allocating the total number of elements required as a one-dimensional array and then passing dimensioning information, along with the array, to a subroutine which accesses the array elements using multidimensional notation. This is illustrated in the following example, in which `IROW` *and* `JCOL`*, the size of each dimension of a two-dimensional array, have been obtained from the user. This example calls* `FALLOC`*, which is defined in the preceding example.*

```
REAL A(1)
POINTER(IPA, A)
INTEGER FALLOC
.
.
.
IPA = FALLOC(4*IROW*JCOL)
CALL MAKEARRAY(A,IROW,JCOL)
.
.
.

SUBROUTINE MAKEARRAY(A,IROW,JCOL)
REAL A(IROW,JCOL)
.
.
.
```

Fortran 90 automatic arrays provide a simpler way to allocate dynamic arrays local to a subroutine. Refer to Appendix C, "Fortran 90 compatibility," for more information on these dynamic memory allocation features.

The `POINTER` *statement is also supported in Cray mode, along with many Cray functions that are designed to allocate and manipulate dynamic memory. Refer to Appendix D, "Cray Fortran compatibility," for more information.*

For more information on calling non-Fortran routines from Fortran, refer to Fortran User's Guide Chapter 4, "Calling conventions."

DIMENSION statement

The DIMENSION statement names arrays and specifies the bounds of the array. It has the following form:

```
DIMENSION arrname([lb:] ub [, ...]) [, arrname(...)]
```

where:

arrname

is the name of the array you are dimensioning. The array is typed according to a type declaration statement (that can either precede or follow the DIMENSION statement) or implicitly. Multiple arrays can be dimensioned in a single DIMENSION statement. *If the array is to be automatic, arrname must be local to a subroutine.*

lb

is the lower dimension bound. This must be an integer value or parameter and must be smaller than *ub*. *lb* defaults to 1.

ub

is the upper dimension bound. This must be an integer value or parameter and must be greater than *lb* if *lb* exists.

For static arrays, *ub* must be a constant. *For allocatable arrays, ub can be a constant, variable or expression.*

If you do not specify a lower bound, the default value is 1, and the upper bound is the number of elements given for the 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 with a colon.

For automatic arrays, ub and/or lb for at least one dimension must either consist of or be derived from an integer argument passed to the subroutine in which the automatic array resides.

Different dimensions of a multidimensional automatic array can be declared to be different sizes based on one or more dimension arguments passed to the subroutine.

The type of array and the product of the subscripts determine the number of storage units allocated to each array named in the statement. Fortran arrays can have from one to seven dimensions.

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

The preceding example specifies an implicitly typed 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. For more information on declaring and dimensioning arrays, refer to Chapter 3, "Arrays and substrings."

ALLOCATABLE statement

Allocatable arrays are dynamic arrays of predetermined rank. Storage for allocatable arrays is allocated on the heap at runtime when an `ALLOCATE` *statement is executed, and must be deallocated through use of the* `DEALLOCATE` *statement when the array is no longer needed. Refer to Chapter 3, "Arrays and substrings," for more information on the* `ALLOCATE` *and* `DEALLOCATE` *statements.*

The `ALLOCATABLE` *statement is used to declare allocatable arrays. Allocatable array declarations have the following form:*

```
[type arrexp]
ALLOCATABLE arrexp [,...]
```

where

type

is an optional type definition for the array. The form for this is identical to the form for a standard array type definition, except instead of specifying constant dimension parameters, you must either supply a rank definition or provide no parameters.

arrexp

is the array name or rank definition if it was not given in a preceding type statement. The array name may occur in both the type and `ALLOCATABLE` *statements; the rank definition must occur in exactly one, or the compiler flags an error.*

An allocatable array's rank must be supplied either in the `ALLOCATABLE` *statement, in a* `DIMENSION` *statement, or in the array's type declaration, which, if used, precedes the* `ALLOCATABLE` *statement. Rank definitions are discussed in detail in the "Allocatable array declarations" section of Chapter 3, "Arrays and substrings." For a more detailed discussion of allocatable arrays, refer to Appendix C, "Fortran 90 compatibility."*

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 contains 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.

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

In this example, variables F, G, and H share the same location; the 9th element in array MYARRAY 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.

Note

Use of the EQUIVALENCE statement can interfere with program optimization.

Equivalencing arrays

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

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)).

The following table shows how these statements align the arrays.

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

Two or more elements of the same array cannot share the same storage location. In the following example, the 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.

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

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

Equivalencing substrings

When making character substrings equivalent for storage, the EQUIVALENCE statement also defines storage locations for the other corresponding characters in the strings.

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

specifies that PROD(6) and N(4), PROD(7) and N(5), and so on through PROD(10) and N(8) share storage locations. Similarly, N(1) now shares storage with PROD(3), N(2) with PROD(4), and so forth.

Equivalencing 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 contiguous storage units are to be noncontiguous.

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.

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

The preceding example 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(5), and B(1,2) through B(2,3); B(1,1) and B(2,1) comprise the extended portion preceding the COMMON block, which is invalid.

STATIC statement

The STATIC statement instructs the compiler to store a specified list of variables in static storage, rather than stack-based storage.

This statement is available only when Sun Fortran compatibility is requested via the -sfc option.

The format of the STATIC statement is as follows:

```
STATIC varlist
```

where

varlist is a list of variables (separated by commas) that will be allocated static storage.

AUTOMATIC statement

The AUTOMATIC statement instructs the compiler to store a specified list of variables in stack-based storage, rather than static storage.

This statement is available only when Sun Fortran compatibility is requested via the -sfc option.

The AUTOMATIC statement uses the following format:

```
AUTOMATIC varlist
```

where

varlist is a list of variables (separated by commas) that will be allocated stack-based storage.

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.

and

varlist

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

An entity can be of type INTEGER, REAL, LOGICAL, COMPLEX, *or* CHARACTER. *If the entity and the constant value assigned to it are not of the same type, the compiler performs the arithmetic assignment conversion. You cannot, however, convert between numeric and character data types.*

The following example shows the format for namelist-directed input:

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

The following statement illustrates the use of NAMELIST:

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

In this example, the NAMELIST *statement defines two group names—*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. (Refer to Chapter 9 and Chapter 10 for more information on namelist-directed I/O.)

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. *It also is used in establishing array-valued functions.* The EXTERNAL statement has the following form:

```
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, all references to the intrinsic name are treated as references to an external procedure rather than as calls to an intrinsic function. 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.

INTRINSIC statement

The INTRINSIC statement permits a name of a specific intrinsic function to be used as an actual argument. Generic intrinsic functions cannot be used in this manner. The INTRINSIC statement has the following form:

```
INTRINSIC intrname [,intrname]...
```

where *intrname* 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.

The following categories of intrinsic functions cannot be used as actual arguments: type conversion (for instance, INT, IFIX, IDINT, REAL, FLOAT, SNGL, DBLE, CMPLX, ICHAR, CHAR), and maximum and minimum value (such as MAX, MAX0, AMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, and MIN1).

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 form:

```
SAVE [n [,n]... ]
```

where *n* is a variable name, statically-sized array name, or a named COMMON block contained between slashes (for example, /NCOM/). Dummy argument names, subprogram or function names, automatic or allocatable array names, or COMMON block entity names are 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.

The SAVE statement implicitly applies to all items that appear in a DATA statement. For instance, the statement

```
DATA Y/4.75/
```

establishes an initial value of 4.75 for Y and forces the value of Y to be saved upon termination of the procedure in which it is declared.

For more information about the DATA statement, refer to Chapter 6, "DATA statement."

DATA statement

6

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

All items listed in a DATA statement are allocated static storage. This means the values of entities that appear in a DATA statement are saved upon termination of the procedure in which they are declared. This is equivalent to the SAVE statement, which is covered in "SAVE statement" section of Chapter 5.

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. Dummy argument names or function names cannot appear 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 two values are associated with 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.

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

This statement initializes a CHARACTER value of NAME for C, LOGICAL value .FALSE. for LOW, and 3 for MYEXAMPLE(2) and MYEXAMPLE(3). As long as you retain the correct name and value association, the order and grouping is immaterial.

The previous example can also be represented as:

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

When a CHARACTER entity is longer than its corresponding CHARACTER constant, blanks are added on the right. If a CHARACTER entity is shorter, extra characters on the right are ignored. For example, the following statements initialize DUR to TEMPORARY^^^ and LOC to HOLDING^CELL.

```
CHARACTER*12 DUR,LOC
DATA DUR,LOC/'TEMPORARY','HOLDING^CELL'/
```

The CHARACTER entity is the same length as the constant in LOC, so no blanks are added or ignored. Three blanks are automatically added to DUR, however, because the CHARACTER entity is longer than its corresponding CHARACTER constant.

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

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

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

Implied-DO

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

(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.

m_1, m_2, m_3

are INTEGER constant expressions that can contain implied-DO variables of other implied-DO lists. The constants specify the initial value, terminal value, and increment, respectively, for the implied-DO variable. If you omit the comma and the value of m_3, the increment value defaults to 1. The increment count must be positive.

Examples using implied-DO:

The statements:

```
REAL C(8), D(12), E, F
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 submatrix 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 includes elements B(1,1), B(2,1) to B(2,2), B(3,1) to B(3,3), and B(4,1) to B(4,4).

DATA statement extensions

If, in the DATA statement, 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 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 has fewer digits than the variable or array element, the constant is extended on the left with zeros. If the constant has more digits than can be stored, the constant is truncated on the left.*
- *If 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 has fewer characters than the variables or array element, spaces are added to the right of the constant. If the constant has more characters than can be stored, excess characters on the right of the constant are eliminated.*

The constant value in clist can 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 PI, the CHARACTER variable C*4 to "'TEST", and the CHARACTER variable CR*1 to the ASCII character code '0D'X.*

```
DATA R, 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. Each variable or array element can only be initialized once.

Assignment statements 7

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

An assignment statement has the form:

v = ex

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

If the type of the variable on the left side of the equal sign is the same as that of the expression on the right, the value is assigned directly. If the data types differ, the value of the expression is converted to the type of the left side entity before the value is assigned. For example, in the statement

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

both the variable I and the expression (L+M)/K are of type INTEGER, so the value 5 is assigned directly. If the variable is INTEGER and the expression is not, the expression is converted to INTEGER and assigned to the variable. Thus, the statement

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

assigns the INTEGER value of 5 (17/3 truncated) to I. In this example, (R+S)/T is truncated and converted to INTEGER.

CHARACTER conversions

The CHARACTER assignment statement has the following 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, only the character positions within the CHARACTER variable or array element are affected. If a value was previously assigned, the value remains the same; if the value was undefined, it remains undefined. Using a differing substring within the same variable, such as A1(1:3) = A1(4:6), is acceptable.

Array assignments

CONVEX Fortran supports several methods of manipulating array data. These include array-valued expressions in assignment statements, masked array assignments (WHERE statements and constructs), array constructors, and vector subscripts. These methods of assigning array data are discussed in this section.

Array-valued expressions

CONVEX Fortran supports Fortran 90 array-valued expressions in assignment statements. This feature allows you to assign a value or expression to an entire array or array section with one assignment, using the following form:

arr = ex

where arr *is the name of an array or an array section description and* ex *is an expression.*

As with assignment to a scalar variable, mismatched expression types are converted to the type of the argument on the left before assignment.

```
INTEGER IX(10)
REAL X
     .
     .
     .
IX = IX * X
```

Here, each element of IX *is multiplied by the* REAL *variable* X *and truncated to an* INTEGER*. The result then replaces the original element in the array.*

In most cases, you can use intrinsic routines in array-valued assignment statements. Most intrinsic routines return array results when passed array arguments.

```
REAL A(15), B(15)
.
.
.
A = SIN(B)
```

In the above example, each element in array A *is assigned the sine of its corresponding element in array* B*. For more information about using intrinsics in array assignments, refer to Appendix A, "Intrinsics and commonly used library routines."*

Note

Fortran 90 array assignments are translated into loops by the compiler. Any optimization options specified at compile time are then applied to the generated loop; for instance, the loop would be parallelized at optimization level *-O3.*

Array sections can be similarly assigned. Refer to the section "Array sections" in Appendix C for more information.

For more general information on the Fortran 90 features available in CONVEX Fortran, refer to Appendix C, "Fortran 90 compatibility."

Masked array assignment

CONVEX Fortran allows the assignment of values to an array under a mask specified via the WHERE *statement or* WHERE

construct. The WHERE statement evaluates a logical expression to determine to which elements the assignment is being applied. The WHERE construct works similarly, but is terminated with the ENDWHERE statement. It can contain several assignments and an ELSEWHERE statement, which allows alternate assignments to be applied for the complement of *mask-expression*.

The WHERE statement has the following form:

```
WHERE(mask-expr) assignment-stmt
```

The WHERE construct has the following form:

```
WHERE(mask-expr)
   [assignment-stmt]
   [...]
[ELSEWHERE
   [assignment-stmt]
   [...]]
ENDWHERE
```

where

mask-expr

is a logical expression of the same shape as the array(s) being manipulated in the *assignment-stmt(s)*.

assignment-stmt

is an array assignment. The array must be the same shape as the array in *mask-expr*.

On execution, the *mask-expr* is evaluated. Any following assignments are executed only on array elements corresponding to those elements for which *mask-expr* evaluated to .TRUE. If an ELSEWHERE statement is present, its assignments are applied to array elements corresponding to those elements for which *mask-expr* evaluated to .FALSE.

```
REAL
DATA(1000),OFFSET(10000),ADJUSTED(1000),LMT
LOGICAL NORMAL(1000)
LMT = 130.0
      .
      .
      .
WHERE(DATA .LE. LMT)  NORMAL = .TRUE.
```

In this example, all elements of the LOGICAL array NORMAL that have the same index as elements in the real array DATA that are less than or equal to LMT are set to .TRUE.

The following example assumes the same variable declarations as Example 1.

```
WHERE(DATA .GT. LMT)
  ADJUSTED = FIX(DATA)
  NORMAL = .FALSE.
ELSEWHERE
  ADJUSTED = 0.0
  NORMAL = .TRUE.
ENDWHERE
```

In this example, elements of ADJUSTED corresponding to elements of DATA greater than 130.0 are set to FIX(DATA), and all other elements of ADJUSTED are set to 0.0. Similarly, all elements of NORMAL corresponding to DATA elements greater than 130.0 are set to .FALSE. and all other elements of NORMAL are set to .TRUE.

Array sections and subscript expressions can be used with the WHERE construct to change the correspondence of elements between arrays.

```
WHERE(DATA .LE. LMT)
  OFFSET(ISET:ISET+1000) = DATA
ELSEWHERE
  OFFSET(ISET:ISET+1000) = 0.0
ENDWHERE
```

In this example, elements of OFFSET starting at the value ISET and going through ISET+1000 are set to correspond to elements of DATA where the DATA elements are less than or equal to LMT, and are set to 0.0 where the DATA elements are greater than LMT.

Note

It is possible to assign values to the same array element in both the .TRUE. and .FALSE. branches of the WHERE construct when using array sections or subscript expressions that differ in each branch of the construct. This action can inhibit vectorization and parallelization of the WHERE construct, and should therefore be avoided.

Remember, the WHERE construct differs from the block-IF in that both clauses can and often do execute. It is therefore possible for assignments that execute in the ELSEWHERE clause of the WHERE construct to change values assigned in the WHERE clause when array section notation is used. The following example, which

illustrates this, assumes the same variable declarations as Example 1.

```
WHERE(DATA .LE. LIMIT)
  OFFSET(ISET:ISET+999:2) = DATA(1:1000:2)
ELSEWHERE
  OFFSET(ISET+1:ISET+1000:2)= OFFSET(ISET:ISET+999:2)
ENDWHERE
```

In this example, every other element from `OFFSET(ISET)` *through* `OFFSET(ISET+999)` *is set in the* `WHERE` *clause. Then in the* `ELSEWHERE` *clause, because of the sectioning notation used, each value assigned in the* `WHERE` *clause is copied into the element immediately following it in* `OFFSET`*. The end result is that* `OFFSET(ISET) = OFFSET(ISET+1) = DATA(1)`*,* `OFFSET(ISET+2) = OFFSET(ISET+3) = DATA(3)`*, and so on for every pair of elements of* `OFFSET` *starting at* `ISET`*.*

Array constructors

CONVEX Fortran supports Fortran 90 array constructors, which can be used to assign values to elements of an array. An array constructor may be a scalar expression, an array expression, an implied-`DO` *specification, or a combination of all three forms.*

Array constructor assignments use the following form:

```
arr = (/ arr_constr_list /)
```

where

> *arr is an array or an array section and arr_constr_list is a list of scalar values, array expressions, or implied-*`DO` *specifications to be assigned to arr. Both arr_constr_list and arr must be of the same type and shape.*

The following example assigns values to elements 1 through 10 of the `REAL` *array* `COST`*:*

```
COST(1:10) = (/ 9.95, SALE(1:7), 2.75, 4.98 /)
```

Following the assignment, `COST(1)` *is* `9.95`*,* `COST(2)` *has the value of* `SALE(1)`*,* `COST(3)` *has the value of* `SALE(2)`*, and so on.*

The implied-`DO` *form of array constructor has the form*

```
( dlist, i = m1, m2 [, m3] )
```

where

dlist

is a list of array element references and implied-DO lists. Array constructors handle up to one level of implied-DO.

i

is the name of an INTEGER variable (the implied-DO variable). The implied-DO variable has the scope of the implied-DO loop. If an array constructor contains more than one implied-DO loop, each implied loop must have its own implied-DO variable.

m1, m2, m3

are INTEGER constant expressions that specify the initial value, terminal value, and increment, respectively, for the implied-DO variable. If you omit m3, the increment defaults to 1. The increment must be positive.

These constants can contain implied-DO variables of other implied-DO lists.

The following assignment uses the implied-DO array constructor:

```
REAL A(10)
A = (/ (SQRT(REAL(I)), I = 0, 27, 3) /)
```

*This statement assigns the square root of the current value of I to all ten elements of array A. Because the SQRT intrinsic accepts only REAL*4 values, the integer I is converted to this type using the REAL intrinsic. A(1) is assigned SQRT(0.0), A(2) is assigned SQRT(3.0), A(3) is SQRT(6.0), and so on.*

The following code is equivalent to the previous implied-DO array constructor:

```
I = 1
DO J = 0, 27, 3
   A(I) = SQRT(REAL(J))
   I = I + 1
END DO
```

Vector subscripts

Vector subscripts provide a way to reference multiple, discontinuous elements of an array. A vector subscript is a rank-one array of INTEGER values used as a subscript to reference multiple elements of an array. All values in a vector subscript must fall within the dimensions of the referenced array. The following example references four elements of array A.

```
INTEGER K(4)
REAL A(10)
...
K = (/ 1, 7, 3, 4 /)
A(K) = 47.0
```

This example sets elements 1 to 4 of array K to be 1, 7, 3, and 4, respectively, using an array constructor. By using a vector subscript (K) to reference array A, elements 1, 7, 3, and 4 of A are assigned a value of 47.0.

Data conversion rules

Table 9 summarizes the data conversion rules for assignment statements. *These rules apply for both arrays and scalar variables of the indicated type.*

Table 9 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.*
	*REAL*16*†	*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.*

Table 9 (continued) Conversion of expressions

Type of variable (V)	Type of expression (E)	Value assigned
REAL	*INTEGER/ LOGICAL*	Convert to REAL and assign to V.
	REAL	Direct assignment of E to V.
	*REAL*8*	*Assign most significant digits of E to V; least significant digits of E rounded.*
	*REAL*16*†	*Assign most 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 most 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.*
	*REAL*16*†	*Assign most significant digits of E to V; least significant digits of E rounded.*
	COMPLEX	Assign real part of E to most significant portion of V; assign 0 to least significant portion of V. Imaginary part not used.
	*COMPLEX*16*	*Assign real part of E to V. Imaginary part not used.*
*REAL*16*†	*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*	*Assign E to most significant portion of V. Least significant portion of V is assigned 0.*
	*REAL*16*†	*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 most significant of V; assign 0 to least significant of V. Imaginary part not used.*

Table 9 (continued) Conversion of expressions

Type of variable (V)	Type of expression (E)	Value assigned
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.*
	*REAL*16*†	*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.*
*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.*
	*REAL*16*†	*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	*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.*

†On CONVEX C Series machines REAL*16 is available only in native mode.

ASSIGN statement

The ASSIGN statement allows you to assign a statement label to an INTEGER variable. It has the following form:

```
ASSIGN s TO i
```

where

s

is the label of an executable statement or FORMAT 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. The variable cannot be referenced in any other way while defined with a statement label value.

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

The above example transfers control to a statement with the label 75. Do not use the ASSIGN statement for arithmetic purposes. For example, `ASSIGN 75 TO M` is not equivalent to `M = 75`. Likewise, an arithmetically assigned variable cannot be used as a statement label. 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, the statement

```
M = 50
```

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

Control statements

8

CONVEX Fortran usually executes statements in the order in which they appear. Control statements provide a means of altering the normal program execution sequence. Control statements include:

- GOTO — Transfers a program's control to a specified statement. Includes unconditional GOTO, computed GOTO, and assigned GOTO statements.
- IF — Transfers control based on the value of an arithmetic or logical expression. Varieties are: arithmetic IF statements, logical IF statements, and block IF statements that may contain ELSE IF, ELSE and END IF statements.
- DO — Provides a mechanism for repeating a section of code a specific number of times. Both indexed DO and *DO WHILE* loops are supported.
- *END DO* — Specifies the end of a DO or a *DO WHILE* loop.
- CONTINUE — Is a neutral, nonexecutable statement. Can be used to mark a GOTO branch point or a DO or *DO WHILE* termination point.
- CALL — Transfers program control to a subroutine.
- RETURN — Returns program control from a subroutine to the calling program unit.
- STOP — Terminates program execution.
- PAUSE — Suspends program execution.
- END — Ends a main program; this statement must end every Fortran program unit.

GOTO statements

You can use 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.

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.

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

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

Computed GOTO statement

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

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

where

slist

is a list of 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 on 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, the arithmetic expression (e) is converted to an INTEGER data type when necessary.

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

Here X is converted to type INTEGER before it is compared to (10, 15, 20, 15, 30) to determine the branch destination.

Assigned GOTO statement

The assigned GOTO statement has the following form:

```
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 statement labels separated by commas ($l_1, l_2,...$) within the program unit.

When an assigned GOTO is executed, *v* must have the value of a label attached to some executable statement in the same program unit. This label should not be attached to a statement that exists within a block IF statement or DO statement block. If several statement label values are present for *slist*, the label assigned to *v* must be a member of that list.

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.

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.

Arithmetic IF statement

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

```
IF (e) l1, l2, l3
```

where

e

is an arithmetic expression.

l_1, l_2, l_3

are labels of executable statements contained within the current program unit. All three labels must be included. Do not use labels of statements within DO statement blocks or IF statement blocks.

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. If the value is equal to zero, control transfers to the statement with the label l_2. If the value is greater than zero, control transfers to the statement with the label l_3. For example, the following statement:

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

transfers control to the statement with label 40 if the product of IA and 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
```

Logical IF statement

The logical IF statement has the form:

```
IF (le) es
```

where

le

is a logical expression.

es

is an executable statement other than a DO, *END DO*, END, logical IF, or 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 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, and SUB1 is not called.

Block IF statement

The block IF statement permits one statement or a block of statements to be executed depending on the value of the logical expression. The block begins with an IF THEN statement, followed by the statement block, and ends 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
```

Where *le* is a logical expression. If *le* 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 include one or more statements in the block.

Consider the following example:

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

If H is less than or equal to 40, 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 following form:

```
IF (le) THEN
     .
     .   !EXECUTABLE STATEMENTS FOR TRUE VALUE
     .
ELSE
     .
     .   !EXECUTABLE STATEMENTS FOR FALSE VALUE
     .
END IF
```

If the logical 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 next executable statement by means of the END IF statement. However, if *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 IF statement is false. For example, in the block

```
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 greater than 40. The ELSE block executes and, unless the ELSE transfers control out of the block, control passes to the END IF, which transfers control to the next executable statement. However, if H is less than or equal to 40, the IF THEN block executes; the ELSE block is skipped.

A more complex block IF statement has the form:

```
IF (le1) THEN
    .
    .
ELSE IF (le2) THEN
    .
    .
ELSE IF (len) THEN
    .
    .
ELSE                    !OPTIONAL
    .
    .
END IF
```

This IF block allows any number of additional logical expressions (*le*) to be specified. If the value of the first logical expression (le_1) is false, the ELSE IF expressions are evaluated until the value of the expression is true. If le_2 through le_n are false, those block sequences are skipped and control transfers to the optional ending ELSE statement, or, if none is present, to 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 the statements associated with that ELSE IF THEN statement execute and control transfers to the END IF statement, which transfers control to the next executable statement. If all ELSE IF THEN statements evaluate to false, the ELSE statement block, if there is one, executes.

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 an IF THEN statement and ends with an END IF statement.

Example:

```
IF (T .GT. 40) THEN
    Y = 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 (Y=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.

Short-circuit evaluation of conditionals

Short-circuiting the evaluation of conditionals increases the efficiency of IF statements by skipping irrelevant tests when LOGICAL operators are involved in the conditional. CONVEX Fortran short-circuits evaluation of IF statements that contain .AND. and .OR. operators which have LOGICAL operands and are used in a logical context. For example, the following IF statement:

```
IF ((A .EQ. B) .OR. F(G)) THEN
```

Assuming `A = B`*, the compiler evaluates* `(A .EQ. B)` *and once it has determined that this condition is true, skips the evaluation of* `F(G)` *and executes the* `THEN` *portion of the statement. Similarly, given the code*

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if `(A .EQ. B)` *evaluates to false, the compiler skips the evaluation of* `F(G)` *and proceeds past the* `THEN` *portion of the statement.*

Short-circuit evaluation works with all types of `IF` *statements (arithmetic, logical, and block). Performing arithmetic (+, -, *, /) on or applying non-*`LOGICAL` *operands or functions to a logical expression disables short circuit evaluation within that expression.*

Logical valued expressions used as arguments to function calls within an `IF` *statement's conditional expression are not short-circuited. Note that the binary operators* `.EQ.`, `.NE.`, `.LE.`, `.GT.`, *and* `.GE.` *always produce a* `LOGICAL` *result.*

The compiler short-circuits evaluation of conditionals by default. You can disable short circuiting by specifying the `-nosc` *option on the compiler command line.*

DO statement

A DO statement specifies a DO loop. A DO loop is a mechanism for repeating a section of code a definite number of times.

A DO loop consists of a DO statement, a group of statements that follow the DO statement and are located within the program unit, and a terminal statement.

You cannot transfer control into the range of a DO loop 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 following 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. *If you do not specify a label, the* `DO` *loop must end with an* `END DO` *statement. That is, a statement of the form* `DO I=1,100,2` *must end with* `END DO`*.* Nested DO loops cannot share an unlabeled `END DO` statement, but they can 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, END, or DO statement as the terminal statement.

v

is the name of the INTEGER, REAL, double-precision, or quad-precision variable called the DO-variable.

ex_1

is an INTEGER, REAL, double-precision, or quad-precision expression that specifies the beginning value of *v* on the initial execution of the DO loop. (Each succeeding value is determined by $ex_1 + ex_3$, $ex_1 + 2\ ex_3$, $ex_1 + 3\ ex_3$).

ex_2

is an INTEGER, REAL, double-precision, or quad-precision expression that specifies the ending value of *v*.

ex_3

is an INTEGER, REAL, double-precision, or quad-precision expression that indicates the increment for *v* after each time the body of the DO loop is executed. If you omit the increment value, it 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, and ex_3 to determine the beginning value of *v*, to determine the final value of *v*, and to set the step value. 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.

Transfer of control outside the loop terminates loop execution. The DO-variable *v* retains its value at loop termination. For example, the following statement:

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

executes the DO loop with MYEXAM taking the values 2, 4, and 6. The loop terminates when MYEXAM becomes 8, and this value remains in MYEXAM after termination.

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.*

Because internal representation of REAL numbers is not exact, using a REAL number for the DO-variable can produce an unexpected count. 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. The actual terminal and increment values used in the loop are not affected either.

After execution of the DO loop (provided the loop is not nested), control transfers to the first executable statement after the terminal statement. If the loop is nested and the loops share a terminal statement, control transfers outward to the next most enclosing DO loop.

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 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.

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

In this example, the DO loops share a common terminal statement at line 10.

Extended-range DO loops

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 following rules apply to extended-range DO *statement control transfers:*

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

The following example illustrates valid control transfer into the range of a 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
```

The following example illustrates invalid transfer into the range of a 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
```

DO WHILE statement

The DO WHILE *statement allows continual execution of a* DO *loop as long as a logical expression contained in the statement remains true. This statement has the following 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 iteration 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.*

Example:

```
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 becomes true during the execution of the loop, the loop does not terminate until control passes back to the top of the loop.

END DO statement

The END DO *statement ends the* DO *and* DO WHILE *statements. This statement may be written either with or without a space (either* END DO *or* ENDDO*).*

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 can also use the* END DO *statement as a labeled terminal statement if the* DO *or* DO WHILE *statement contains a terminal-statement label.*

```
   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
```

CONTINUE statement

Because the execution of a CONTINUE statement has no effect, you can place it anywhere in a program that an executable statement is allowed.

When used as a terminal statement in a DO loop, a CONTINUE statement must be labeled; otherwise, no label is required. This statement has the following form:

```
[s]   CONTINUE
```

where *s* is an optional statement label.

In the following example, the CONTINUE statement is used as a GOTO branch point.

```
    ...
30  CONTINUE
    ...
    IF (BVAL .GT. AVAL) GOTO 30
```

The CONTINUE statement can be used in a similar way to mark the end of a DO loop:

```
    ...
    DO 60 K = 1, 125, 5
       TC = TC + A(K)
       ...
60  CONTINUE
```

CALL statement

The CALL statement transfers program control to a subroutine. It has the following form:

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

where *sub* is the name of the subroutine and *a* is an actual argument or argument list. Refer to Chapter 11, "Subprograms," for more information.

RETURN statement

The RETURN statement returns control from a subroutine to the calling program unit. It has the following form:

```
RETURN [e]
```

where *e* is an optional integer expression that specifies an alternate statement in the calling program that is to receive

control. Refer to Chapter 11, "Subprograms," for a complete description.

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'
```

PAUSE statement

The PAUSE statement suspends program execution until the operator orders execution to resume. It has the following form:

```
PAUSE [s]
```

where

s

is a string of five or fewer digits or a character constant to be printed.

Example:

```
PAUSE 'LOAD TAPE NUMBER 1'
```

When the preceding statement is executed, the system displays the following information:

```
PAUSE:  LOAD TAPE NUMBER 1
To resume execution, type: go
Any other input will terminate the program.
```

To continue, type go and press **RETURN**. The system displays:

```
Execution resumed after PAUSE.
```

END statement

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

```
END
```

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

Input/output statements

9

Input/output (I/O) statements provide a method for transferring data between internal storage and external media or between internal storage and internal files. This chapter describes the statements and methods CONVEX Fortran supports for I/O on both C Series and SPP Series machines.

Overview of I/O

CONVEX Fortran supports formatted, list-directed, namelist-directed, and unformatted I/O. Formatted I/O statements have explicit format specifiers that control data translation from internal binary form within a program to external, readable-character form in the records, or vice versa.

List-directed and namelist-directed I/O statements, although similar to the formatted statements used in functions, use data types rather than explicit format specifiers to control data translation from one form to another.

Unformatted (or binary) I/O statements do not translate the data being transferred and can be used when output data is later to be used as input. Unformatted I/O saves execution time; it eliminates the translation process, maintains greater precision in the external data, and conserves file storage space.

I/O statements transfer all data as records. The amount of data a record can hold depends on whether unformatted or formatted I/O is used for data transfer. With unformatted I/O, the I/O statement determines how much data is to be transferred. With formatted I/O, the I/O statement and its associated format specifier determine how much data is to be transferred.

Usually, data transferred by an I/O statement is read from or written to a single record. However, a formatted, list-directed, or namelist-directed I/O statement can transfer more than one record.

C Series and SPP Series I/O differences

CONVEX Fortran I/O support differs slightly on C Series and SPP Series machines. SPP Series Fortran provides more compatibility with Hewlett-Packard I/O than does C Series Fortran. For HP compatibility information, see Appendix F, "HP Fortran compatibility."

The main difference between SPP Series and C Series I/O is in the assignment of implicit unit numbers to `stderr`. See Table 12 and Table 13 on page 100 for this information.

Supported I/O statements

CONVEX Fortran supports READ, *ACCEPT*, *DECODE*, and *BUFFER IN* statements for input. The WRITE, PRINT, *TYPE*, *ENCODE*, and *BUFFER OUT* statements are accepted for output. Auxiliary statements control the connection of files to external devices, position files, or retrieve information about a file or unit. The auxiliary statements CONVEX Fortran supports are OPEN, CLOSE, REWIND, INQUIRE, BACKSPACE, ENDFILE, and *FIND*. The *FIND* statement is available only on C Series machines.

Table 10 lists the I/O statements CONVEX Fortran supports by category.

Table 10 Input/output statements

Type	Statement	Use
Input	READ	Transfers data from an external file into internal storage or between internal storage locations.
	ACCEPT	Sequentially reads data from the standard input unit.
	DECODE	Transfers data between arrays or variables in internal storage and translates the data from character to internal form.
	BUFFER IN	Reads data while allowing unrelated subsequent processing to proceed concurrently. For more information refer to Appendix D, "Cray Fortran compatibility," of the *Fortran Language Reference*.
Output	WRITE	Transfers data from internal storage to an external device or between internal storage locations.
	PRINT	Transfers formatted records to the standard output device.
	TYPE	Same as PRINT.

Table 10 (continued) Input/output statements

Type	Statement	Use
Output (continued)	*ENCODE*	Transfers data between arrays or variables in internal storage and translates the data from internal to character form.
	BUFFER OUT	Writes data while allowing unrelated subsequent processing to proceed concurrently. For more information refer to Appendix D, "Cray Fortran compatibility," of the *Fortran Language Reference*.
Auxiliary	OPEN	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.
	CLOSE	Disconnects a file from a unit.
	REWIND	Positions a file at its initial point.
	INQUIRE	Determines the specified properties of a file or of a unit on which a file can be opened.
	BACKSPACE	Positions a file to the preceding record.
	ENDFILE	Writes an endfile record on the file connected to the specified unit.
	FIND	Positions a direct-access file to a particular record. Available only on CONVEX C Series machines.

Supported I/O methods

Table 11 summarizes the I/O methods CONVEX Fortran supports and the I/O statements permitted for use with each method when accessing files of various types. The information presented here is elaborated upon throughout this chapter.

Table 11 Input/output methods

File type and I/O method	I/O statement						
	READ	WRITE	ACCEPT	TYPE	PRINT	DECODE	ENCODE
Internal/ Sequential:							
Formatted	Yes	Yes	*No*	*No*	No	*Yes*	*Yes*
List-directed	*Yes*	*Yes*	*No*	*No*	*No*	*No*	*No*

Table 11 **(continued)** Input/output methods

File type and I/O method	I/O statement						
	READ	WRITE	ACCEPT	TYPE	PRINT	DECODE	ENCODE
External Sequential:							
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*
External/ Direct:							
Formatted	Yes	Yes	*No*	*No*	No	*No*	*No*
Unformatted	Yes	Yes	*No*	*No*	No	*No*	*No*

Unformatted internal I/O statements, direct list-directed, and direct namelist-directed I/O statements are not allowed. All other variations are allowed.

Records, files, and units

This section discusses how to manage records, files, and units when writing CONVEX Fortran code that performs I/O.

A record is a sequence of characters or values processed as a single collection. I/O statements transfer data in the form of a record. A file is a sequence of records that are input to or output from a program. Accessing an external file requires connecting the file with a unit.

Records

A sequence of characters or values processed as a unit constitutes a record. I/O statements transfer all data as records.

Records are either formatted or unformatted. Formatted records contain characters and are read and written with format specifications. Unformatted records (those written without format specification) consist of bytes that represent binary values and have header and trailer fields containing the record length.

Each unformatted I/O statement transfers one record. Formatted, list-directed, and namelist-directed I/O statements transfer as many records as required by the I/O data list. Each read or write starts a new record.

Formatted records

A formatted record contains a sequence of characters (letters, numbers, and special symbols). Formatted records can be read or written only by formatted input/output statements. You cannot use formatted I/O on files connected for unformatted access.

With formatted input, if the input statement requires more characters than are available, characters are read as spaces. If the input statement does not require all the characters in the record, unneeded characters 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. When the size of the data is greater than the record length and when an output statement writes to a fixed-length record, an error condition occurs.

Unformatted records

An unformatted record is a sequence of zero or more bytes, surrounded by a four-byte header and a four-byte trailer, each containing the record length. 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 bytes 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 bytes left unfilled during data transfer to fixed-length records become zeros.

`ENDFILE` record

The `ENDFILE` statement writes the `ENDFILE` record that ends the file. An `ENDFILE` record is also written when a file opened for writing is closed, either through the `CLOSE` statement, through a `REWIND` statement, or implicitly through program termination.

The `ENDFILE` record appears only as the last record of a file. When such a file is closed with a `REWIND` statement, the `ENDFILE` record is written at the current position before rewinding. You cannot use an `ENDFILE` statement on a file connected for direct access.

Files

A sequence of records that are input to or output from a program constitutes a file. There are two types of files: external and internal. An external file is associated with a disk file, terminal, or some other device. An internal file is associated with internal storage space and consists of a character variable, array element, array, or substring.

Files can be accessed in two ways: sequentially and directly. File access is covered in the "Accessing files" section later in this chapter.

For details about the maximum permissible file size under the operating system you use, refer to *Fortran User's Guide* Appendix G. File size limits due to non-operating system factors are noted where appropriate.

Internal files

An internal file is 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. Internal files provide transfer and conversion of data from internal storage to internal storage.

A record in an internal file can be read only if the record is defined. When the processor writes a 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.

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 is the unit 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 can 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` (on C Series machines) or `ftn07` (on SPP Series machines); this is the file associated with unit 7 by default. The following statement:

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

connects unit 8 to the file `TEST.DAT`.

The following statement:

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

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

To reassign a 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.

When an `OPEN` statement is executed for an unopened unit, the program environment is searched for a shell variable associated with the unit. This variable is named `FORnnnOPEN`, where nnn is a three-digit number representing the unit (for example, from 000 to 999, leading zeroes required). This shell variable can contain attributes that override the attributes specified in the `OPEN` statement for that unit. Refer to the "Conversion using a shell variable" section of this chapter for more information and examples.

In the absence of an associated shell variable, a unit takes its attributes from the list of attributes specified with the `OPEN` statement. Attributes not specified in the shell variable are also taken from the `OPEN` statement.

Every unit in CONVEX Fortran, except `stderr` (unit 0 on C Series machines, unit 7 on SPP Series), is associated, by default, with a logical name that is used to create a default actual file name. On C Series machines this name has the form `fort.`*nnn*, where *nnn* is the unit number. On SPP Series machines this name takes the form `ftn`*xxx*, where *xxx* is the unit number. For information about logical file names, refer to the "Logical file names" section of this chapter.

Implicit unit numbers

In certain forms of the READ and WRITE statements and in statements such as ACCEPT, PRINT, or TYPE, the unit number is implicit. Table 14 on page 101 shows the general forms of CONVEX Fortran I/O statements in which the unit is specified implicitly.

C Series only

Table 12 shows the default C Series unit numbers CONVEX Fortran assigns to stdin, stdout, and stderr. stdin is used for input, stdout for output, and stderr for error messages. For a summary of SPP Series defaults see Table 13.

Table 12 Implicit Fortran unit numbers (C Series only)

File	Default unit number
stdin	5
stdout	6
stderr	0

SPP Series only

Table 13 lists the default SPP Series unit numbers CONVEX Fortran assigns to stdin for input, stdout for output, and stderr for error messages.

Table 13 Implicit Fortran unit numbers (SPP Series only)

File	Default unit number
stdin	5
stdout	6
stderr	7

On both C Series and SPP Series machines, when you use an asterisk (*) in a READ or WRITE statement, unit 5 or 6 is always referenced, regardless of whether or not the unit has been specified in a CLOSE or OPEN statement.

All three of these designators are normally assigned to your terminal. You can redirect units 5 and 6 with operating system commands or with the OPEN statement. You can reopen units 5, 6 and stderr (either unit 0 on C Series machines, or unit 7 on SPP Series), but the C language pointers stdin, stdout and stderr are not reassigned.

Table 14 lists the implicit unit numbers used on both C Series and SPP Series machines when the unit is specified implicitly by the following I/O statements. In this table, *f* indicates the FORMAT statement number and *list* indicates the data to be transferred.

Table 14 Implicit units numbers by Fortran statement

Fortran statement	Implicit unit
READ (*, *f*) *list*	5
READ *f*, *list*	5
ACCEPT *f*, *list*	5
WRITE (*, *f*) *list*	6
PRINT *f*, *list*	6
TYPE *f*, *list*	6

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.

Sequential access

To connect a file for sequential access, use the OPEN statement.

Examples:

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

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

A file connected for sequential access cannot be read or 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 order of placement in the file. The last record must be an endfile record.

Direct access

Connecting a file for direct access allows the records to be written or read in any order. The record number specified in the I/O transfer statement determines the order of processing. You cannot use sequential access I/O statements on files connected for direct access.

To establish a direct-access file, open a unit for direct access.

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. Every time you read or write a record, you must specify a record number to indicate the record to be read or written.

I/O statement format

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

```
READ (clist) iolist
WRITE (clist) iolist
```

where

clist

is the control information list that controls the data transfer.

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.

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, for example, `M(1,1)`, `M(2,1)`, and so on.

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:

$(dlist, v=ae_1, ae_2 [, ae_3])$

where

dlist

is an I/O list.

v

is an integer or real variable.

ae_1, ae_2, ae_3

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 ae_1 and increments by the value of ae_3 until it equals or exceeds the value of ae_2. 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)
C     WRITES THE PAIR A,B 10 TIMES

      READ (7)(A(I),I=5,10)
C     READS ELEMENTS 5 THROUGH 10 OF ARRAY A

      WRITE (7)((A(I,J),J=1,N),I=1,N)
C     WRITES THE ARRAY A BY ROWS
```

Specifiers

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

- Unit — Identifies the unit being accessed.
- Format — Specifies the type of formatting.
- Record — Indicates which record is to be accessed (used only with direct access files).
- Status — Provides a means for determining an error or end-of-file condition.
- Error — Designates a statement to receive control if an error occurs during program execution.
- End of file — Designates a statement to receive control if an end-of-file condition is detected.
- *Namelist — Specifies that namelist-directed I/O is to be used and to specify the group name.*

Unit specifier

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

```
[UNIT= ] u
```

where *u* is an internal or external identifier. As an external file identifier, *u* is an integer in the range 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.

Format specifier

You must include a format specifier in each data transfer statement to or from a formatted file. A format specifier is the 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.

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 can not use the record specifier for sequentially accessed files.

A record specifier has the following 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. While *r* can be any size INTEGER, the value of *r* must fit in an INTEGER*4 storage location. *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.*

Note

Because the record number for I/O must be accessible through an INTEGER*4 variable assigned via the REC specifier in an I/O statement, the number of records in the file cannot exceed 2^{31}-1 records. On C Series machines, this may override the normal one terabyte-512 bytes file size limit under ConvexOS. Refer to the *ConvexOS Extensions User's Guide* for more information. On SPP Series machines this may override the file size limit under SPP-UX.

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 one of the following:

- A 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.

Error specifier

An error specifier designates a statement to receive control if an error occurs during program execution. An error specifier has the following 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.

End-of-file specifier

You can use the end-of-file specifier in a statement 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.

Namelist specifier

The namelist specifier specifies that namelist-directed I/O is to be used and specifies the group name for the entities that are modified during input or written on output. The namelist specifier has the following 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 use the keyword NML =. You cannot use the namelist specifier in a statement containing a format specifier.

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 following form:

```
READ (clist) [iolist]
```

or

```
READ f [,iolist]
```

where

clist

is a control information list (described in the "Specifiers" section of this chapter). 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 files. 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, or array names.

f

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

When the READ statement executes, at least one record consisting of values from the I/O list is read. The file is then positioned at the beginning of the next record.

By default, CONVEX Fortran reads a sequential, formatted file unless an OPEN statement contains FORM='UNFORMATTED' or ACCESS='DIRECT'. Unformatted reads from internal files are not permitted.

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

External sequential-access READ statements

There are four classes of external sequential-access 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.

Formatted

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]
```

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.

Unformatted

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

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

Example:

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

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

List-directed

A list-directed sequential-access statement, which must contain an asterisk (*) to indicate list-directed formatting, has the following forms:

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

Examples:

```
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.

Namelist-directed

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 without specifying a control information list, it has the following 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.

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, and 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, and EXAM3) in the order they appear in the input records. (See Chapter 10, "Format specifications," for detailed information on inputting values.)*

External direct-access READ statements

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

Formatted

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

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

Example:

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

This statement uses the format given at line 100 to read record number 25 of the file connected to unit 119 and assigns values to variables D, E, and F from this record.

The REC and END keywords are mutually exclusive. The following statement is invalid:

```
READ(10,20,REC=1,END=30) A      ! Wrong!
```

Unformatted

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

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

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.

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.

There are two types of internal READ statements: sequential-access and direct-access.

Sequential access

The internal sequential-access READ statement is always formatted and has the following 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
```

Direct-access

The internal direct-access READ statement has the following form:

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

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.

ACCEPT statement

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

```
ACCEPT f [,iolist]
```

or

```
ACCEPT* [,iolist]
```

or

```
ACCEPT nlgrpname
```

where

f

is the non-keyword form of a format specifier.

*

specifies list-directed formatting.

iolist

is an I/O list.

nlgrpname

is the non-keyword form of the namelist specifier.

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

Example:

```
          ACCEPT 100, I, J
100       FORMAT (2I2)
```

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

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,f) [iolist]
```

where

clist

is a control information list (described in the "Specifiers" section of this chapter) that must include a unit specifier. A formatted record must include a format specifier. A record specifier must be included for direct-access output to a file. Status and error specifiers are optional. (An end-of-file specifier is not allowed in WRITE statements.)

iolist

is the I/O list that identifies the data to be transferred. The entities can include variables, array elements, substrings, implied DO lists 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. In the absence of an OPEN statement, the type of file written by CONVEX Fortran is dependent on the form of the WRITE statement used. If an OPEN statement is present and specifies FORM='UNFORMATTED' or ACCESS='DIRECT', an unformatted file is written. 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 follows the same rules as formatted I/O.

A WRITE statement of the following form normally writes to `stdout` (for example, to the terminal):

```
WRITE (*, *) [iolist]
```

In a FORTRAN 77 subroutine called from a C program, however, this statement writes to the file `fort.6` by default. To change this behavior, you must either initialize Fortran I/O by calling `f_init` from the C program, or define the environment variable FOR006 as SYS$OUTPUT.

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.

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.

Formatted

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

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

Examples:

```
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 variables D, E, and F to the file connected to unit 50. 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.

Unformatted

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

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

Examples:

```
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.

List-directed

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

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

Example:

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

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

Namelist-directed

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

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

Example:

```
WRITE (UNIT=50, NML=SAMPLE)
```

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

External 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.

Formatted

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

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

Examples:

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

Both statements write variables `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.

Unformatted

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]
```

Examples:

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

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

Both statements write variables 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.

Internal WRITE statements

The internal WRITE statement 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.

There are two classes of internal WRITE statements: sequential access and direct-access.

Sequential access

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

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

Examples:

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

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

Direct access

The internal direct-access WRITE statement has the form:

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

Example:

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

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

PRINT and TYPE statements

You can use either the PRINT statement or the *TYPE* statement to transfer formatted records to the standard output device. These statements use the sequential mode of access and have the following forms:

```
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 non-keyword form of the namelist specifier.

Example:

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

The TYPE statement writes one record to the standard output device; the record contains four fields of character data.

Special input/output statements

The *ENCODE*, *DECODE*, and *FIND* statements are extensions to the ANSI standard and have been included to allow for compatibility with other versions of Fortran and for ease in transporting older Fortran programs to CONVEX machines. The *FIND* statement is available only on C Series machines.

ENCODE statement

The ENCODE statement is equivalent to the internal formatted sequential-access WRITE statement. ENCODE transfers data between arrays or variables in internal storage and translates the data from internal to character form.

The ENCODE *statement has the following form:*

```
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 are padded 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 that of 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'
```

In this example, the character string A *gets the contents of the integers* I *and* J.

DECODE statement

The DECODE *statement is equivalent to the internal sequential-access* READ *statement.* DECODE *transfers data between arrays or variables in internal storage and translates the data from character to internal form. The* DECODE *statement has the following form:*

```
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 that 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 statement determines the number of characters that the* DECODE *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 that of 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
```

In this example, the contents of the character string A *are transferred to the integers* I *and* J.

FIND statement (C Series only)

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 available only on C Series machines and 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`*.*

Auxiliary input/output statements

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

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

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.

ConvexOS allows a Fortran program to have a maximum of 256 files open at one time, including standard input (stdin), standard output (stdout), and standard error (stderr).

On SPP-UX, a program may have a maximum of 60 files open at once, including the stdin, stdout, and stderr files.

The OPEN statement has the following form:

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

Where *specifier* is an expression normally including a keyword and its value. You must include a unit number in the OPEN statement; all other specifiers are optional. The specifiers can be listed in any order except that the unit number must be first when it is given without the UNIT= keyword.

An OPEN statement without a FILE keyword opens the default file for that unit unless STATUS='SCRATCH' is specified. If STATUS='SCRATCH' is specified, a temporary file is generated with the name tmp.F*cppppppnnn*, where *c* is a special character, *ppppp* is the process ID and *nnn* is the unit number. By default, a temporary file is deleted when the program ends.

The following example opens a file named fort.94 in the current home directory.

```
OPEN (94)
```

The following example opens a temporary file, tmp.F*cppppppnnn*, in the current working directory and deletes it when program execution is complete:

```
OPEN (24, STATUS='SCRATCH')
```

The following example opens the specified file:

```
OPEN (10, FILE='/usr/tmp/myfile')
```

This example 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:

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

The following sections describe the OPEN statement keywords, which are summarized in Table 15. In the descriptions, the term "numeric expression" can be any integer or real expression. The value of the expression is converted to the INTEGER data type before it is used in the OPEN statement.

Table 15 OPEN statement keywords

Keyword	Values	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' *'APPEND'*	Access mode	'SEQUENTIAL'
ASSOCIATEVARIABLE (C Series only)	*V*	*Next direct-access record*	*None—must specify value*
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 Series only)	*C*	*Default file specification*	*None—must specify value*
DISPOSE or DISP	*'KEEP' or* *'SAVE' or* *'DELETE'*	*File disposition at close*	*Depends on STATUS keyword*
ERR	S	Error transfer label	None—must specify value
FILE *or NAME*	C	File-name specification	fort.*n* (C Series) or ftn*n* (SPP Series), where *n* is the unit number
FORM	'FORMATTED' *'PRINT'* 'UNFORMATTED' *'UNFORMATTED/ dataformat'*	Format type	'FORMATTED' for sequential access; 'UNFORMATTED' for direct access

Table 15 (continued) OPEN statement keywords

Keyword	Values	Function	Default
IOSTAT	*V*	*I/O status*	*None—must specify value*
MAXREC	*E*	*Direct-access record limit*	*Unlimited*
NOSPANBLOCKS	*None allowed*	*Ignored—for VAX compatibility only*	*None—must specify value*
READONLY (C Series only)	*None allowed*	*Write protection*	*Depends on file access rights*
RECL *or* *RECORDSIZE*	E	Record length	As specified at file creation
RECORDTYPE or *RT*	*'FIXED'* *'VARIABLE'*	*Record structure*	*'VARIABLE' for sequential access;* *'FIXED' for direct access*
STATUS *or* *TYPE*	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
UNIT	E	Logical unit number	*None—must specify value*

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.

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')
```

ASSOCIATEVARIABLE keyword (C Series only)

The ASSOCIATEVARIABLE keyword specifies an integer variable to be updated after each direct access I/O operation. This keyword is available only on C Series machines and 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.

Because the ASSOCIATEVARIABLE is modified by direct-access I/O operations in any routine called by the program in which it is associated, passing it as a parameter to the called routine or declaring it in a COMMON memory area can create a hidden alias. Avoid passing the ASSOCIATEVARIABLE or declaring it in a COMMON memory area.

Note

Optimizing routines that reference the ASSOCIATEVARIABLE can cause unexpected results. Always compile these routines at optimization level -no.

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.

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 64 kbytes. For other devices (such as raw tape), the BLOCKSIZE value is rounded up to a multiple of the file system block size. An INQUIRE statement with a BLOCKSIZE keyword returns the rounded-up value.

The following statements write one physical record of 200 bytes to the block-mode tape device /dev/mt12. The physical record contains 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)
            .
            .
            .
```

CARRIAGECONTROL keyword

The CARRIAGECONTROL keyword determines the carriage control processing to be used for 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. 'LIST' transmits the first character of each formatted output record unchanged. Fortran replaces the first character of each formatted output record with the control characters required to interpret it on an ASCII output device. These control characters are ***CTRL-L*** *for start of page, newline for double spacing, and null (no character) for single spacing.*

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

DEFAULTFILE keyword (C Series only)

The DEFAULTFILE keyword defines part of a default file specification. This keyword is available only on C Series machines and has the form:

```
DEFAULTFILE = c
```

where c is a character expression.

The DEFAULTFILE keyword is used to fill in missing parts of the file name specified with the FILE= or NAME= keyword.

```
      OPEN(FILE='info',DEFAULTFILE='.dat')
C     OPEN FILE INFO.DAT
```

This statement opens the file info.dat.

```
   OPEN(FILE='info.dat',DEFAULTFILE='/mnt/user1/')
C  OPEN /MNT/USER1/INFO.DAT
```

This statement opens the file /mnt/user1/info.dat. Under the COVUEshell, the equivalent statement would appear as in the following example.

```
   OPEN(FILE='info.dat',DEFAULTFILE='mnt:[user1]')
C  OPEN MNT:[USER1]INFO.DAT
```

The DEFAULTFILE keyword is used mainly for interactively requesting a file name, especially to fill in a part of the file name that is a default, such as a directory name or extension. Any components in the FILE= keyword override those in the DEFAULTFILE= keyword.

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:

file is opened for unformatted sequential access, the file is in the proper format. By default, this test is disabled. When enabled, this test checks files opened with FORM='UNFORMATTED' and either ACCESS='SEQUENTIAL' or ACCESS='APPEND'.

SPP Series only

The unformatted sequential access test is enabled by declaring a COMMON block named IOLIB__CHECKUF and placing within it a non-zero INTEGER*8 value. For example:

```
INTEGER*8 ENABLE
COMMON/IOLIB__CHECKUF/ENABLE
DATA ENABLE/1/
```

When this test is enabled and a file opened for unformatted sequential access is not in the correct format, the runtime error message "903 unformatted I/O attempted on formatted file" is generated. This same message is produced when an unformatted READ or WRITE is attempted on a formatted file.

When the unformatted sequential access test is enabled, it is performed from program startup until disabled. To disable this test within a program, assign the COMMON block's INTEGER*8 variable a value of zero using an executable statement, for example:

```
ENABLE = 0
```

When Cray Fortran compatibility is enabled (via the -cfc option) this test should *not* be enabled for already existing files that are created with BUFFEROUT statements and are opened as unformatted sequential access files.

IOSTAT keyword

The IOSTAT keyword provides a variable that is set to indicate the status of an 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.

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)
```

NOSPANBLOCKS keyword

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

```
NOSPANBLOCKS
```

READONLY keyword (C Series only)

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. This keyword is available only on CONVEX C Series machines.

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')
```

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 shown in Table 16.

Table 16 RECORDTYPE defaults

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)
```

SHARED keyword

The SHARED keyword is provided for compatibility with Fortran source written for non-UNIX operating systems only. ConvexOS, like UNIX, provides for shared files by default. The compiler ignores the SHARED keyword.

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 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.

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. Valid logical unit numbers are 0 through 255.

If the unit number appears as the first parameter of the OPEN statement, the UNIT keyword can be omitted; otherwise, 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 which the unit is currently connected is opened. 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 file to which the unit is connected, the effect is the same as executing a CLOSE statement immediately prior to executing the OPEN statement.

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

CLOSE statement

Use the CLOSE statement to disconnect a file from a unit. The CLOSE statement has the following forms:

```
CLOSE ([UNIT=]u [,STATUS=p][,ERR=s][,IOSTAT= ios])
```

or

```
CLOSE ([UNIT=]u [,DISPOSE=p][,ERR=s][,IOSTAT=ios])
```

or

```
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 following 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 supersedes 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.

INQUIRE statement

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

```
INQUIRE (FILE=fi, list)
```

or

```
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.

Note

On CONVEX C Series machines, the FILE name you specify when using the INQUIRE statement under the COVUEshell must be the absolute ConvexOS path name. VMS path names are not recognized.

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 17 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 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 17 enumerates and describes INQUIRE specifiers.

Note

Because the next record number must be accessible through an INTEGER*4 variable assigned via the NEXTREC specifier in an INQUIRE statement, the number of records in the file cannot exceed 2^{31}-1 records. On C Series machines, this may override the normal one terabyte-512 bytes file size limit under ConvexOS. Refer to the *ConvexOS Extensions User's Guide* for more information.On SPP Series machines this may override the file size limit under SPP-UX.

Table 17 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 of 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.
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**	*file name* if file has name; blank if no file name. If the file is not currently open, the absolute pathname is returned.
NAMED = *logical**	.TRUE. if file has a name; .FALSE. if no name.
NEXTREC = *integer**	*next record number* if record length specified on OPEN.
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.

File-positioning statements

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.

File-positioning statements have the following form:

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

or

```
REWIND u
```

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

or

```
BACKSPACE u
```

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

or

```
ENDFILE u
```

where

u

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

s

is the statement label to which control transfers if an error condition exists. (If 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.)

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
      .
      .
      .
```

BACKSPACE statement

The BACKSPACE statement positions the file connected to the specified unit to 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. Do not attempt to BACKSPACE in an unformatted sequential access Cray pure data file.

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
```

ENDFILE statement

The ENDFILE statement writes an endfile record to 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)
```

Logical file names

Every unit in CONVEX Fortran, except stderr (unit 0 on C Series machines, unit 7 on SPP Series), is associated, by default, with a logical name. Logical names take the form FOR*nnn*, where *nnn* is the unit number. This logical name is used to create a default actual file name in the form fort.*nnn* on C Series machines, or ftn*nnn* on SPP Series machines. These default file names are automatically preconnected to the unit. Preconnected means that the file is connected to the unit when the program begins executing and can be referenced by input/output statements without prior execution of an OPEN statement.

The following statement opens and writes to the file fort.35 (on C Series machines) or ftn35 (on SPP Series machines):

```
WRITE (35,10) data
```

Table 18 shows examples of units, the logical names associated with the units by default, and the actual file names to which they are preconnected by default.

Table 18 Examples of default logical names

Unit	Default logical name	C Series default file name	SPP Series default file name
8	FOR008	fort.8	ftn08
52	FOR052	fort.52	ftn52
230	FOR230	fort.230	ftn230

Units 5 and 6 (stdin and stdout) are not automatically preconnected to files. You must use explicit CLOSE and OPEN statements to get this connection as shown in the following example. In the following example, because no file name is specified in the OPEN statement, the default file name (fort.5 on C Series, ftn05 on SPP Series) is assigned.

Example:

```
CLOSE (5)
OPEN (5)    ! Unit 5 (stdin) is now connected
            ! to fort.5
```

You can usually override the preconnection of a unit with an explicit OPEN statement or with environment variables as described later in this section. If, however, the logical name specified in an OPEN statement is of the form FOR*nnn*, the file actually generated has the corresponding `fort.`*nnn* (C Series) or ftn*nnn* (SPP Series) name.

The following statement generates a file named `fort.4` (on C Series machines) or a file named `ftn04` (on SPP Series).

```
OPEN (UNIT=1, FILE='FOR004')
```

The files `stderr`, `stdin`, and `stdout` can also be redirected from the command line as described in the description of the C shell (`csh`) in the online man pages.

Assigning logical names

You can customize your ConvexOS or SPP-UX environment to assign Fortran logical names to files. You also can change the default name assigned to scratch files by changing the environment variable FORTEMP. For a discussion of your operating system's environment, refer to the CONVEX man pages.

In the most commonly used command interpreter, the C shell (`csh`), the commands `setenv` and `unsetenv` control the setting and resetting of variables in your working environment. These commands have the following format:

`setenv` *name value*

`unsetenv` *name*

To examine the environment variables that are currently set, use the `csh` command `printenv`.

When a unit is opened, the logical name associated with the unit is compared to the environment variables. The logical name can be specified in the FILE= clause of an OPEN statement or can be the default logical name associated with the unit. If an environment variable matches the logical name, the value

assigned to that variable is substituted for the logical name and the comparison process is repeated.

Examples

In the examples in this section, *c* is a special character, *ppppp* is the process ID number, and *nnn* is the unit number.

The following command causes the compiler to generate scratch file names in the current directory of the form `MYFILE`*cppppppnnn* rather than `tmp.F`*cppppppnnn*:

```
setenv FORTEMP MYFILE
```

The following command causes the compiler to generate scratch file names in the directory `/tmp` of the form `TEMP`*cppppppnnn* instead of `tmp.F`*cppppppnnn*:

```
setenv FORTEMP /tmp/TEMP
```

The following example causes data to be written to the file `output.dat` in the home directory:

```
setenv FOR021 ~/output.dat
      .
      .
      .
WRITE (21,*) A, B, C
```

The following example causes data to be written to the file `/acct/smith/data`:

```
setenv FOR055 OUTPUT
setenv OUTPUT /acct/smith/data
      .
      .
      .
WRITE (55,*) BASELINE
```

The following example writes data to `stdout`. You must use the backslash (\) as an escape character:

```
setenv FOR021 SYS\$OUTPUT
      .
      .
      .
WRITE (21,10) IOLIST
```

The following example causes data to be written to the file /acct/smith/printfile:

```
setenv PRINT /acct/smith/printfile
      .
      .
      .
OPEN (21,FILE='PRINT')
WRITE (21,50) I, J, A
```

The following example causes data to be written to ./PRINT:

```
unsetenv  PRINT
      .
      .
      .
OPEN (21,FILE='PRINT')
WRITE (21,50) I, J, A
```

Accessing file pointers

The library routine for$getfp returns the file pointer associated with a Fortran unit number. This routine can be used when it is necessary to modify certain attributes of the file; for instance, it can be used to bypass the buffer cache.

Typically, the file pointer is passed to a C-language routine that actually performs the operation. Note that the C language stdio.h macro fileno() returns a file descriptor given a file pointer.

Example:

Fortran program:

```
INTEGER for$getfp, fp
      .
      .
      .
OPEN (1, file="file1")
fp = for$getfp(1)
call bypasscache(fp)
      .
      .
      .
```

C routine:

```
#include <stdio.h>
#include <fcntl.h>
void bypasscache_(fp_ptr) /* on C Series a
                             trailing
                             underscore is
                             required because
                             the routine is
                             called by a
                             Fortran program
                             (*not* required
                             on SPP Series)  */

  FILE* *fp_ptr;          /* note fp_ptr is a
                             pointer to a
                             FILE pointer
                             (FILE*)        */
{
int fd, n;
FILE* fp;
   fp = *fp_ptr;
   fd = fileno(fp);
   n = fcntl(fd, F_GETFD, 0);
   fcntl(fd, F_SETFD, n | _FNCACHE);
   return;
}
      .
      .
      .
```

Binary data file format conversion (C Series only)

The binary data file format conversion feature of CONVEX C Series Fortran allows programs to read from and write to unformatted, binary data files whose data format is different from the program's mode. Mode refers to the format in which the program is processing data, either `NATIVE` *or* `IEEE`*. For example, a CONVEX Fortran program in native mode can read from and write to an unformatted binary file whose format is* `vax-g`*.*

Note

The material discussed in this section applies only to CONVEX Fortran on C Series machines. This section does not apply to SPP Series Fortran.

CONVEX Fortran allows conversions of the data formats listed in Table 19.

Note

On C Series machines, the REAL*16 data type is only supported in native floating point mode; programs using IEEE mode on C Series machines can neither read nor write REAL*16 data.

Table 19 Data format conversion routine names

Format	Data format names
CONVEX native	`convex_native`, `convex-native`, or `native`
*CONVEX IEEE**	`convex_ieee`, `convex-ieee`, or `ieee`
VAX D_floating	`vax_d` or `vax-d`
VAX E_floating	`vax_d`, `vax-d`, `vax_g` or `vax-g`
VAX G_floating	`vax_g` or `vax-g`
VAX H_floating	`vax_d`, `vax-d`, `vax_g` or `vax-g`
Cray†	`CRAY`
Cray unformatted unblocked sequential access	`CRAYUB`
User-Defined	`user_defined` or `user-defined`

*On C Series machines *REAL*16* is supported in native floating point mode only.
†See Appendix D for details on converting Cray files.

Note

You must use the `cvbin` utility to convert VAX files into a format recognizable by CONVEX Fortran before attempting to read these files. `cvbin` can be used to copy and convert files from a DECnet or COVUEnet node, or it can be used to convert local files. Refer to the cvbin(1) man page or to the *CONVEX COVUEbinary User's Guide* for more information.

You convert data by specifying one of the data format names listed above in an `OPEN` statement or by using a shell variable. Both methods are described in this section. When you specify a data format that is different from the program's current mode, a data conversion routine converts the data when `READ` or `WRITE` statements execute.

For more information on reading and writing Cray files, refer to Appendix D, "Cray Fortran compatibility." For more information

on reading and writing VAX Fortran files, refer to Appendix E, "VAX Fortran compatibility."

When to use the conversion feature

Specify conversion only in programs that match one of the following criteria:

- *"One-time" programs or programs run infrequently.*
- *Programs that read or write small amounts of binary data.*
- *Programs whose data files contain record layouts that vary from record to record (if writing a conversion program is not practical or cost-effective).*

Do not use the conversion feature on programs that repeatedly read unchanging data in the same data file over and over again. A custom conversion program converts data permanently, but the conversion feature must convert data each time it is read.

Programs that read very large files can use a standalone conversion program that is optimized for its particular data file format. This method should be more efficient if data values are known to fall in certain ranges and if issues such as overflow and underflow can be ignored.

`-dfc` option

If you plan to use the format conversion feature with VAX data, then you must also use the `-dfc` *compiler option. The* `-dfc` *option helps convert VAX data by instructing the compiler to decompose the VAX record I/O element by element.*

Conversion using `OPEN` statement

To convert data using the `OPEN` *statement, use the* `FORM` *keyword to specify unformatted data and a data format type:*

```
OPEN (..., FORM='UNFORMATTED/format' , ...)
```

where format is one of the data format names listed in Table 19 and indicates the current data format of the file you want to convert. You can specify the data format in uppercase, lowercase, or mixed case.

When binary data is read from the file specified in the `OPEN` *statement, the conversion routines convert the data from the*

specified data format to the data format implied by the program's mode, either `NATIVE` *or* `IEEE`.

When binary data is written to the file specified in the `OPEN` *statement, the conversion routines convert the data from the data format implied by the program's mode to the data format specified in the* `OPEN` *statement.*

Restrictions on conversions

Observe the following restrictions when using data file format conversions:

- *If a program which uses* `EQUIVALENCE` *statements writes or reads a file specifying any type of conversion and the data type of the actual value in memory does not match the data type of the variable specified in the I/O statement, the value usually becomes corrupted.*
- *For unions in records, the compiler cannot determine the data type of the actual value in memory. Therefore, when a record containing a union is specified in an I/O statement and data format conversion (*`-dfc` *option) is specified, the compiler issues a diagnostic message. You can modify the program to read or write each structure's elements rather than the entire structure.*

  This restriction prevents you from reading or writing data that is probably corrupted.
- *All conversions performed expect* `REAL` *data types to contain numeric data. Using* `REAL` *or* `COMPLEX` *data types for Hollerith data does NOT work if a floating-point conversion takes place.*
- *The VAX does not support* `INTEGER*8` *or* `LOGICAL*8` *data types, so attempting to read or write these data types when the data format is* `vax-d` *or* `vax-g` *causes a diagnostic message to be issued at runtime (unless an* `ERR=` *or* `IOSTAT=` *clause is specified).*
- *On C Series machines* `REAL*16` *is supported only in native mode. Native mode C Series programs that attempt to read or write a* `REAL*16` *value from or to a file opened with data format* `IEEE` *print a diagnostic message during execution and halt (unless an* `ERR=` *or* `IOSTAT=` *expression is specified in the I/O statement).*

Note

Because on C Series machines `REAL*16` variables cannot be specified in `IEEE` mode programs, there is no way to read or write `vax-h` or `NATIVE` format `REAL*16` values in `IEEE` mode C Series programs.

- *The Cray Fortran compiler does not support arithmetic items less than eight bytes long. So attempting to read or write these data types when the data format is* `CRAY` *or* `CRAYUB` *causes a diagnostic message to be issued at runtime (unless an* `ERR=` *or* `IOSTAT=` *clause is specified).*
- *CONVEX Fortran's internal representation of floating point data is different from Cray's; the CONVEX representation allows for greater precision through a slightly smaller range of numbers. If you attempt to read a floating point number written on a Cray and the number is larger than the largest number representable under CONVEX Fortran, the read will fail with a floating point exception.*
- *When the* `FORM = UNFORMATTED/CRAY` *data format conversion is specified, unformatted, direct access, unblocked ("pure") files can be read.*

  By using the `fcUnblock` *utility to convert sequential access, unformatted, blocked files into readable unblocked format, you can access files of this type by specifying* `FORM = UNFORMATTED/CRAY` *data format conversion. Refer to Appendix D, "Cray Fortran compatibility," for details on reading various unformatted Cray files. Refer to the fcUnblock(3f) man page for more information on* `fcUnblock`.
- *Optimizing code containing type conversions (including implicit type conversions) can cause some code to vectorize poorly or not at all.*

Error handling using data format conversions

When you specify `ERR=` *or* `IOSTAT=` *in an I/O statement, most errors reading* `REAL*16` *values in* `IEEE` *mode on C Series machines do NOT cause a diagnostic message to be issued (except for calling stub routines for user-defined I/O). Instead, the program branches to the user-defined label, or if only an* `IOSTAT=` *expression was specified, it branches to the statement immediately following the active I/O statement.*

If you do not specify `ERR=` *and* `IOSTAT=` *in the active I/O statement, then all errors except overflow and underflow cause a diagnostic message to be issued, and execution terminates.*

When an overflow or underflow occurs, then an appropriate value is stored in the user's variable for a `READ` *or in the I/O buffer for a* `WRITE`. *The library then activates an overflow or underflow. If you ignore underflow and overflow, program execution continues. If not ignored, these conditions cause the program to*

receive a signal (SIGFPE) that terminates the program unless trapped via the `signal` routine. You can call `errtrap` to change the default handling of overflows and underflows. By default, programs ignore underflow (zero is substituted), and overflow causes a diagnostic message to be issued and terminates execution.

Reading or writing a reserved operand value (ROP) or Not a Number value (NaN) has no effect on the program's execution. The appropriate value for the target data format is generated. No traps or signals occur.

User-defined conversions

You can create your own conversion routines to convert a data file from a format that does not have a supplied conversion routine. To use your own routine, specify `user_defined` as the data format of the file in the OPEN statement or in a shell variable.

*You must supply two functions for each Fortran data type: REAL*4, INTEGER*2, and so on. One function converts the data from the user-defined format to the appropriate CONVEX Fortran data type. The other function converts a CONVEX Fortran data type to the user-defined data type.*

Table 20 shows a set of stub routines for user-defined data types. You only need to supply the routines that you actually use. If you forget to write a routine that is called, the stub routine prints an appropriate diagnostic message and halts execution of the program. This error message ignores ERR= and IOSTAT= clauses.

Each user-defined conversion routine is called as follows:

```
NAME (source, target, bytecount)
```

where source and target are pointers to a block of storage, and bytecount is a pointer to an integer value. The routine is expected to pick up the values through the source pointer and store the converted data values through the target pointer. bytecount is the number of data values the routine converts multiplied by the size of one item in bytes. For example, a call of the form

```
CVT_INT4_TO_UD ( SOURCE , TARGET , 40 )
```

*converts 10 INTEGER*4 values (40/4) from CONVEX binary integer format to the user-defined format. SOURCE points to the 10 values before conversion, and the user-supplied routine stores the values through TARGET.*

The function must return 0 to indicate success and -1 to indicate failure. The function must also provide support for reserved values, infinity, overflows, and underflows as needed.

Sample conversion routine

The Fortran source code shown in Figure 2 converts `INTEGER*4` *binary values into one's complement binary format. No provision is made in this routine for the maximum negative integer that cannot be represented in one's complement format.*

Figure 2 *Fortran example conversion routine*

```
      INTEGER FUNCTION CVT_INTEGER4_TO_UD (SOURCE, TARGET, BYTECOUNT)<D>
      INTEGER SOURCE(*), TARGET(*), BYTECOUNT
      INTEGER N
      N = BYTECOUNT / 4                    ! COMPUTE AN ITEM COUNT
      DO 10 I=1,N                          ! CONVERT N ITEMS
        TARGET(I) = SOURCE(I)              ! IF VALUE < 0, SUBTRACT 1
      IF (TARGET(I) .LT. 0) TARGET(I) = TARGET(I) - 1
10     CONTINUE
      CVT_INTEGER4_TO_UD = 0               ! RETURN SUCCESS
      RETURN
      END
```

The C source code to accomplish the same task is shown in Figure 3.

Figure 3 *C example conversion routine*

```
int cvt_int_to_ud_ (source, target, byteCount)
   int * source, * target, * byteCount;
 {
   int n;
   n = ( *byteCount) / 4;   /* Compute an item count                 */
   while ( n-- > 0) {       /* Convert n items                       */
        *target = *source;  /* Move the value                        */
        if (*target < 0)    /* If the value is negative,             */
            (*target)--     /* Subtract one.                         */
        source++;           /* Adjust the source & target pointers   */
        target++;           /* for the next value/destination        */
  }
   return (0);              /* Return success.                       */
}
```

Note

A trailing underscore is required on the routine name if it is written in C.

Because some of the Fortran data types are not directly supported in C, it can be useful to declare the pointer's source *and* target *as something else, such as a* `char*` *or a* `struct` *pointer.*

Note

Only one user-defined data format is supported; therefore, a given program is restricted to one set of user-supplied conversion routines.

You can support several different user-defined data formats concurrently by modifying the program to save the current user-defined data format in a variable in COMMON*, which can be queried by each conversion routine.*

User-supplied conversion routine names

Table 20 lists user-defined conversion routine names.

Table 20 User-supplied conversion routine names

Fortran type	Routine names	
	Fortran-to-user-defined	User-defined-to-Fortran
*INTEGER*1*	*cvt_integer1_to_ud*	*cvt_ud_to_integer1*
*INTEGER*2*	*cvt_integer2_to_ud*	*cvt_ud_to_integer2*
*INTEGER*4*	*cvt_integer4_to_ud*	*cvt_ud_to_integer4*
*INTEGER*8*	*cvt_integer8_to_ud*	*cvt_ud_to_integer8*
*REAL*4*	*cvt_real4_native_to_ud* *cvt_real4_ieee_to_ud*	*cvt_ud_to_real4_native* *cvt_ud_to_real4_ieee*
*REAL*8*	*cvt_real8_native_to_ud* *cvt_real8_ieee_to_ud*	*cvt_ud_to_real8_native* *cvt_ud_to_real8_ieee*
*REAL*16*†	*cvt_real16_native_to_ud* *cvt_real16_ieee_to_ud*	*cvt_ud_to_real16_native* *cvt_ud_to_real16_ieee*
*COMPLEX*8*	*cvt_complex8_native_to_ud* *cvt_complex8_ieee_to_ud*	*cvt_ud_to_complex8_native* *cvt_ud_to_complex8_ieee*
*COMPLEX*16*	*cvt_complex16_native_to_ud* *cvt_complex16_ieee_to_ud*	*cvt_ud_to_complex16_native* *cvt_ud_to_complex16_ieee*
*LOGICAL*1*	*cvt_logical1_to_ud*	*cvt_ud_to_logical1*
*LOGICAL*2*	*cvt_logical2_to_ud*	*cvt_ud_to_logical2*
*LOGICAL*4*	*cvt_logical4_to_ud*	*cvt_ud_to_logical4*
*LOGICAL*8*	*cvt_logical8_to_ud*	*cvt_ud_to_logical8*
CHARACTER	*cvt_character_to_ud*	*cvt_ud_to_character*

†On C Series machines *REAL*16* is not supported in IEEE mode programs.

For all floating point data types (REAL, COMPLEX, and so on), the program's mode (NATIVE or IEEE) has been included as part of the name. This allows the runtime library to trap an unintended conversion.

*COMPLEX data types are really two adjacent REAL values. For COMPLEX*8, there are two adjacent REAL*4 values. Because a byte count is passed to the conversion routines, the conversion routines for COMPLEX*8 can usually be identical to the REAL*4 routines.*

Conversion using a shell variable

Each time an OPEN *statement is executed for a unit number not previously opened, the program's environment is searched for the shell variable named* FORnnnOPEN*, where* nnn *is the unit number.* nnn *must be 3 digits long with leading zeros as needed. If this shell variable exists, the attributes it contains are applied to the file in question. If* FORnnnOPEN *is not found, the file attributes specified in the* OPEN *statement are used.*

Any attributes not specified in the shell variable are taken from the OPEN *statement. In the absence of an associated shell variable, the attributes specified with the* OPEN *statement are used.*

All OPEN *operations for that particular unit use those attributes specified in the shell variable.*

Note

***The data format attribute (dataformat=xxxx)* is only valid if the file is opened for unformatted I/O. Otherwise, it is ignored.**

Table 21 lists the attributes that you can specify in the shell variable.

Table 21 Shell variable attributes

Keywords & abbreviations	Legal values
BLANK *BLNK*	*null, zero*
BLOCKSIZE *BLKSZ*	*<number>*
CARRIAGECONTROL	*Fortran, LIST, NONE*
MAXREC	*<number>*
RECL	*<number>*
RECORDTYPE *RT*	*FIXED, VARIABLE*
DISPOSE *DISP*	*KEEP, DELETE*
POSITION *POS*	*as is, REWIND, APPEND*
DATAFORMAT *DF*	*NATIVE, IEEE,* *vax-d, vax-g,* *CRAY, CRAYUB,* *user-defined*

Note

When the data format attribute is specified, either in an `OPEN` statement or a shell variable, the source code must be compiled with CONVEX Fortran V6.0 or higher and linked with the CONVEX Fortran V6.0 or higher libraries. If a version of CONVEX Fortran prior to V6.0 is used, unpredictable behavior, including program aborts and errors, can result.

Use the following `csh` command to specify a `FORnnnOPEN` value, where `nnn` is a three-digit number from 000 to 255.

```
$ SETENV FORnnnOPEN "RECL=256, DATAFORMAT=VAX-G"
```

The corresponding `sh` commands are as follows:

```
$ FORnnnOPEN="RECL=256, DATAFORMAT=VAX-G"

$ EXPORT FORnnnOPEN
```

The corresponding COVUEshell command is:

```
$SET COVUE ENVIRON : FORnnnOPEN = "RECL=256, DATAFORMAT=VAX-G"
```

You can use keyword abbreviations to reduce the total length of the shell variable's value. Keywords and their abbreviations can be given in either uppercase or lowercase. The shell variable name must be in uppercase.

Format specifications 10

Format specifications describe the format of data to be read or written and define any editing that is required. You can use the following methods to specify formats in formatted input/output (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
50 FORMAT (I4)
```

- 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 I/O, for example:

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

FORMAT statement

The nonexecutable FORMAT statement provides 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, D, F, E, G, A, L, and Q.

ned

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

fs

is a nonempty *flist*.

A repeatable edit descriptor has one of the following forms:

[*r*]*c*	[*r*]*cw.d*[*Ee*]
[*r*]*cw*	[*r*]*cw.d*[*De*]
[*r*]*cw.m*	[*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 or D

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 use with INTEGER format descriptors, for example, I, O, Z. Do not use PARAMETER constants for the terms *r*, *w*, *m*, *d*, or *e*.

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 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.

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

This statement 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 newline 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.

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)
```

Descriptors

Field and edit descriptors in CONVEX Fortran are grouped into the categories that follow.

- CHARACTER descriptor: A
- Editing descriptors, character constants, and Hollerith constants:

apostrophe (')	asterisk (*)	T
TL	TR	P
Q	dollar sign ($)	colon (:)
slash (/)	X	H
B, BN, and BZ	S, SP, SS, and *SU*	*R*

- INTEGER descriptors: I, *O*, *Z*
- LOGICAL descriptor: L
- REAL and COMPLEX descriptors: D, E, F, G

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 equals 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 22.

Table 22 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
*REAL*16*	*16*
COMPLEX	8
*COMPLEX*16(DOUBLE COMPLEX)*	*16*

If *w* 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 *w* is greater than the size of the *iolist* item, on input the rightmost characters are stored in the variable.

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. If *w* is less than the number of characters in the *iolist* item, only the leftmost *w* characters are written.

The following example illustrates reading into a CHARACTER*5 variable.

Format code	External field	Internal value
A	CONVEXCOMPUTER	CONVE
A4	CONVEXCOMPUTER	CONV^
A14	CONVEXCOMPUTER	PUTER

The following example illustrates 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

Apostrophe (') descriptor

The apostrophe descriptor has the form of a character constant. 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 statements

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

produce

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

Asterisk (*) descriptor

CONVEX Fortran supports the asterisk descriptor as part of its Cray compatibility. The asterisk descriptor delimits literal text and behaves identically to the apostrophe descriptor. Use two consecutive asterisks to produce a single asterisk.

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 following 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 statements

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

produce

```
ENTER 'FILE' NAME
```

L descriptor

The L descriptor formats LOGICAL variables and has the form:

L*w*

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

The input field is composed of optional blanks, optionally followed by a T (t, .T., .t.) for true or an F (f, .F., .f.) for false. The T or F can be followed by additional characters, for example, .TRUE. or .FALSE., in the field. On input, a field containing only blanks 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 Examples:

Format code	External field	Internal value
L2	T60	.TRUE.
L7	^^FALSE	.FALSE.
L7	1234567	Error: invalid

Output Examples:

Format code	Internal field	External field
L1	.TRUE.	T
L3	.FALSE.	^^F

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 that the field to be edited is *w* characters wide.

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 only 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 is padded with asterisks (*). If you specify *m*, the external field contains up to *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* equals 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 sign (+) does not precede a positive integer. You must include a 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: decimal

Output Examples:

Format code	Internal value	External field
I4	760	^760
I8.4	760	^^^^0760
I3	-760	***
I4	0	^^^0
I4.0	0	^^^^

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 wide.

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 contains fewer digits than m, the remaining positions are padded with zeros on the left. You can only use the numerals 0 through 7 in the external field; you cannot use a decimal point (.), a sign (+ or -), or an exponent field.

Input Examples:

Format code	***External field***	***Internal octal***
O3	523	523
O4	23176	2317
O4	2.317	*Error: decimal*
O4	-1234	*Error: signed*

Output examples:

Format code	***Internal decimal value***	***External value***
O6	4095	^^7777
O6	-4095	*****
O3	4095	***
O4.3	8	^010

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 wide.*

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 has fewer digits than m, the remaining positions are padded with zeros on the left. You can use only the numerals 0 through 9 and the letters* A *(*a*) through* F *(*f*) in the external field; you cannot use a decimal point (.), a sign (+ or -), or an exponent field.*

Input examples:

Format code	***External field***	***Internal hex value***
Z3	9A1	9A1
Z3	9A1B	9A1
Z3	9A.1	*Error: decimal*

Output examples:

Format code	***Internal decimal value***	***External value***
Z4	4095	^fff
Z5	-1	*****
Z6.4	4095	^^0fff
Z2	4096	**

F descriptor

The F descriptor provides formatting of REAL numbers. It has the following form:

F*w.d*

where

w

is an unsigned, positive INTEGER constant that specifies the field to be edited is *w* characters wide.

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 has a decimal point, the *d* term has no effect; 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 only blanks, the value is zero. The processor treats a plus sign (+) or no sign as a positive value; a 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 a 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, for example, at least equal to or greater than *d* + 3. Leading spaces are added, if needed, to fill the field.

Input examples:

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	123.45678

Output examples:

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 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.

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*, E*w.d.e*, or E*w.d*E*e*
D*w.d*, D*w.d.e*, or D*w.d*E*e*

where

w

is the width of the field containing the REAL number. The width is the count of all characters in the field, including sign (if any), decimal point, and exponent.

d

is the fractional part of the field that contains *d* digits.

E or D

identifies the exponent part that contains *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, not including the exponent, 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, a two- or three-digit exponent (depending on whether the I/O list item is REAL*4 or REAL*8), E or D, and the sign of the exponent.

All data values, including *REAL*16*, can be used with the E and D formats.

In a data value, the exponent can be a signed INTEGER constant, or E or D followed by zero or more blanks, followed by an optional signed INTEGER constant. You can use the legal characters—digits 0 through 9, decimal point, plus, minus, E, D, and blank. With the descriptors in the form of E*w.d*, *Ew.d.e* or E*w.d*E*e* (D*w.d*, D*w.d.e*, *Dw.d*E*e*), the value of the next item in the list has the following form:

$$[\pm][0].x_1,x_2 \ \ldots \ x_d \ exp$$

where

signifies a plus or minus sign; the plus sign is optional for a positive value.

0

an optional leading zero.

$x_1, x_2 \ldots x_d$

are the *d* most significant digits of the value after rounding.

exp

is a decimal exponent that is of the form $E \text{—} z_1[z_2] \ldots z_{e3}$ where z is a digit and there are *e* digits in the exponent.

Input examples:

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 examples:

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	*****
E7.2.1	2.718	0.27E+1

If the value is too large for the field, asterisks (*) are output. The default length for the exponent field for *REAL*16* numbers is three. (On C Series machines the REAL*16 data type is available only in native mode.) In native format, if the sign is 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.

G descriptor

The G descriptor edits external single-precision, double-precision, quad-precision, or COMPLEX data. It has the following form:

```
Gw.d, Gw.d.e or Gw.dEe
```

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 significant digits to be printed. On output, when the range of the value to be printed forces E style editing (see below), *d* specifies the number of characters to the right of the decimal point. When the range of an output value allows F editing, *d* specifies the number of significant digits to print.

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 can 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 uses either the F or E style of editing 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 greater than or equal to `10**d`, the output editing of *Gw.d* or *Gw.dEe* is the same as that of *k*P*Ew.d* and *k*P*Ew.dEe*, respectively, and *k* is the scale factor currently in effect. If M is greater than or equal to 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 shown in Table 23.

Table 23 Data conversion based on magnitude

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 G*w.d* and *e* + 2 for G*w.d*E*e*. 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 an (*e* + 2)-character exponent. Thus, you must make *w* equal to or greater than *d* + 7 or *d* + 5 + *e*.

Input examples:

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. The default length for the exponent field for *REAL*16* numbers is three. In native format, if the sign is 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.

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 supersede the default interpretation of blanks (spaces and tabs) 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, blanks can either be interpreted as zeros or ignored, depending on the value of the BLANK specifier in the OPEN statement. The BLANK specifier can be omitted from the OPEN statement, and the OPEN statement itself can be omitted.

Additionally, the `-vfc` compiler option and use of the COVUEshell can effect the default value of BLANK. COVUEshell is available only for C Series machines. Table 24 shows the value of the BLANK specifier under various definition states.

Table 24 BLANK specifier defaults

OPEN/BLANK definition		BLANK specifier value	
OPEN statement	BLANK specifier	Under COVUEshell or -vfc	-vfc and COVUEshell not used
Omitted	NA	BLANK = 'ZERO'	BLANK = 'NULL'
Included	Omitted	BLANK = 'NULL'	BLANK = 'NULL'
Included	Included	Specified value	Specified value

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 the BN specifier, 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 containing only 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 NULL value or the BN descriptor is specified, the statements:

```
READ (5,'(I4,I7,I6)') L, M, N
READ (5,'(BN,I5,I6,I6)') L, M, N
```

assign the value of 5 to L, the value of -500 to M, and the value of 3056 to N. The processor ignores all blanks. If the BZ descriptor is indicated by:

```
READ (5,'(BZ,I5,I6,I6)') L, M, N
```

the values assigned become L = 50; M = -50000; N = 30056. The processor treats all nonleading blanks as zeros. If another input statement refers to unit 5, blank interpretation returns to the default value.

The descriptor B causes a 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:

```
READ (5,'(BZ,I5,I6,B,I6)') L, M, N
```

The value of 3056 is assigned to N rather than 30056, because the B descriptor returns blank interpretation to default mode.

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:

nP

where *n* is an optional INTEGER constant that specifies the number of positions, to the left or right, that the decimal point is to be moved. The value can be signed. Its default value is 0.

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 (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
`-1PF7.3`	`58.967`	`^^5.897`
`2PE10.3`	`890.11`	`^89.01E+01`

If you use a scale factor when an external field has an explicit exponent, for example, `5.E02`, the processor ignores it; in this case, `500` is stored regardless of the scale factor.

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 an `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 descriptors. *The SU descriptor only affects `INTEGER` values.* The descriptors have the following forms:

S

reverts to normal interpretation.

SP

adds a plus sign (+).

SS

suppresses plus signs.

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 sign characters from any position where the plus sign is optional. 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 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)
```

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 $2 \leq n \leq 36$. 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.

Example:

```
      I = 15
      WRITE (6,10) I, I, I
10    FORMAT (16R,I4,8R,I4,R,I4)
```

produces

```
^^^F^^17^^15.
```

X descriptor

The X descriptor sets the position in a record and has the form:

```
nX
```

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 that 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 two blanks before 450 and three blanks before 8921.

The X format descriptor cannot in itself change the length of a record. For example, the statements

```
      WRITE (6,100) I
100   FORMAT (I4,X)
```

produce a four-character record followed by a newline, not a trailing blank.

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

```
Tn
```

where *n* specifies the absolute position within the record. This form indicates transmission of characters at position *n*. For example, if a file connected to the designated input unit contains the record:

```
^2.5^200^^40
```

then execution of the 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 statements

```
      PRINT 50
50    FORMAT (T10,'MY',T13,'EXAMPLE')
```

output MY at position 10 and EXAMPLE at position 13.

Another form of the T descriptor is

T or nT

which causes tabbing to the next (or nth) 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)
```

cause 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 following 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 third T-series descriptor has the following 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 and X field descriptors are identical. For example, assume the external record is as follows:

```
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` descriptor cannot itself change the record length. For example, the statements

```
      WRITE (6,10) I
10    FORMAT (I4,TR10)
```

produce a 4-character record with no trailing blanks.

Dollar sign ($) descriptor

The `$` descriptor suppresses the newline at the end of the current record of a formatted sequential-access write. (In an input statement, the `$` descriptor is ignored.) For terminal I/O, a typed response follows the output on the same line. For example, the 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.

Q descriptor

The `Q` descriptor determines the number of unread characters in the current record. It is represented by:

```
Q
```

Example:

```
      READ(4,100) J,MYEXAM,(ISAM(I),I=1,MYEXAM)
100   FORMAT(I5,Q,80AI)
```

This example reads the first field into variable `J`, 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.

Colon (:) descriptor

The colon (:) descriptor ends format control when no items remain in the I/O list. If items remain in the I/O list, the colon descriptor has no effect. The following 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.

```
    M = 15
    N = 500
    WRITE (10,40) M, N
40  FORMAT (I2,:,' SAMPLE',I4)
```

writes 15 SAMPLE 500; the colon descriptor is ignored because items remain in the I/O list.

Slash (/) descriptor

The slash (/) descriptor indicates the end of data transfer for the current record. For example, the statements:

```
    READ (10,50) L, M, N
50  FORMAT (I2/I4,I3)
```

cause L to be read from the first record, and 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 the 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.

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. These default values are shown in Table 25. On CONVEX C Series machines the REAL*16 data type is available only in native mode.

Table 25 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		
O, Z	*REAL*16*	44		
L	LOGICAL	2		
F, E, G, D	REAL, COMPLEX*8	15	7	2
F, E, G, D	*REAL*8, COMPLEX*16*	24	15	3
F, E, G, D	*REAL*16*	42	33	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	*REAL*16*	16		
A	CHARACTER*n	n		

Comma field separator

A comma between numeric fields overrides the width specified in the field descriptor. Because you can use a comma to end 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 end a field that is controlled by an A or a 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.*

Runtime formats

Format specifications, called runtime formats, can be stored in CHARACTER variables, character substrings, and character expressions, and in character arrays, *numeric arrays and numeric array elements. 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.

Example:

```
INTEGER*8 IFMT
CHARACTER*8 SFMT
IFMT = 8H(2X,I12)
SFMT = '(2X,I12)'
WRITE (6,IFMT) I
WRITE (6,SFMT) I
```

Cray-style asterisk descriptors can be used in runtime formats in all modes.

Variable formats

A variable format has an expression, enclosed in angle brackets, that is computed 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 statement is subject to the following rules:

- If the expression is not an INTEGER, it is converted to an 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; a runtime error will occur.

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)
    3 FORMAT(1X,'Message = ',A<LEN(STRING)>)
      J = 2
      READ(5,1) A,B,C,D,E,F
      WRITE(6,2) A,B,C,D,E,F
      WRITE(*,3) STRING
      STOP
      END
```

Note that variable formats are *not* allowed in runtime format strings, as shown in the following example.

```
WRITE(*,'(1X,"Message = ",A<LEN(STRING)>)')!
INVALID!!
```

Attempting this will result in a runtime error.

List-directed formatting

List-directed formatting transfers data based on the data type of the entity. A list-directed I/O statement uses 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 is a sequence of values and value separators. A value is generally a constant but can also be a null value, or the value can have the form

*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 value). 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 part of a 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

List-directed 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 to a character constant; likewise, a numeric element must correspond to 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 (refer to Chapter 7, "Assignment statements"). 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.

Character input

Character constants for list-directed input are usually enclosed in apostrophes. Character constants can span record boundaries.

Embedded blanks, commas, and slashes within a character string are not considered separators. 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, or blank (space or tab). A newline ends the string unless escaped with a slash (/). Any string not meeting these restrictions must be enclosed in single or double quotes.

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.

A null value 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 ends the READ statement. Those items skipped retain their original values.

Namelist-directed input 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

$

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.

For the NAMELIST statement

```
NAMELIST/SAM/ NAME, EXAM1, EXAM2, EXAM3
```

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 $
```

Several acceptable formats for entering input exist. For example, you can also enter the previous input as

```
$SAM^NAME='TESTA'^EXAM1=5.2^EXAM2=6.78^EXAM3=10.0
$END
```

You can use commas, tabs, and spaces as valid separators in the list of value assignments, and input can begin in any column. You cannot use nonblank control characters in column 1.

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, and so on. The number of array elements you can assign must be less than or equal to the size of the array.

Assume a program with the following statements:

```
DIMENSION MYRAY(10)
NAMELIST /SAM2/ MYRAY
READ SAM2
```

Assume that the input is

```
$SAM2 MYRAY = 10, 8, , 70 $END
```

On 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 because the two consecutive commas in the input indicate not to change the current value, and values for unspecified array elements remain unchanged.*

Because values are assigned to the specified array elements and not assigned 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 following rules for list-directed input:

- *Do not use spaces or tabs in groupname definitions. In value assignments, the entity name cannot contain spaces or tabs except within a subscript or substring specifier, where they 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 equal 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 can 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 reads and writes records of a fixed length.*

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, neither double nor single quotation marks are output for character constants. Table 26 shows the default output forms that the list-directed WRITE statement generates for each data type.

Table 26 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
*REAL*16*	*1PG43.33E4*
COMPLEX	1X,'(',1PG14.7E2,',',1PG14.7E2,')'
*COMPLEX*16*	*1X,'(',1PG23.15E3,',',1PG23.15E3,')'*
CHARACTER	A*n*, where *n* represents the character expression length

Namelist-directed output formatting

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, assume a program with the following statements:

```
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 the following:*

```
$CONTROL
L4          = F,
I4          = ^^^^-123213,
R4          = ^^3.141590^^^^,
C8          = (^^12.00000^^^^,^^2.000000^^^^),
CHAR20      = 'test^case^^^^^^^^^^'
$END
```

The output for this program segment consists of 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 not assigned an input value. (The entity can 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 as well as the 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 mentioned above, the data types used are the same as those defined in the `NAMELIST` *statement. The format output follows the same form as that of list-directed output.*

Although you can use the $ and & characters interchangeably on input, the $ character is always used for output.

Carriage-control characters

The first character of a formatted record transfers to the printer as a carriage-control character. Table 27 shows the characters that provide vertical format control.

Table 27 Vertical format control

Character	Interpretation
^	Advances one line; begins output at beginning of next line.
0	Advances two lines; skips one line and begins output.
1	Advances 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.*

Specifying `FORM='PRINT'` in the `OPEN` statement indicates formatted I/O and implies vertical format control for that unit. You can use the `fpr` utility to interpret the vertical format controls before printing the file.

Subprograms 11

Subprograms are program units that can be invoked from other program units. Subprograms usually perform frequently used sequences of operations for the invoking program unit. A subprogram's arguments (both dummy and actual arguments) 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. BLOCK DATA subprograms provide initial data for COMMON block variables and arrays. Procedure subprograms include functions and subroutines.

BLOCK DATA subprogram

A BLOCK DATA subprogram provides initial values for variables and array elements in named COMMON blocks. Unlike procedure subprograms, BLOCK DATA subprograms cannot be called by the programmer; instead they are invoked once at program execution to establish initial values for variables and array elements. A BLOCK DATA subprogram must not contain any executable statements.

A BLOCK DATA subprogram begins with a BLOCK DATA statement and ends with an END 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 that of an external procedure, main program, COMMON block, or other BLOCK DATA subprogram, or 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
- any type-declaration statements

Your BLOCK DATA subprogram must contain at least one COMMON statement and one DATA statement.

You must specify all entities having storage units in the COMMON block, but you are not required to initialize all of the values. Be sure to provide specifications to establish the entire block.

Example:

```
BLOCK DATA MYBLOK
COMMON /EX/ A,B,C
COMMON /CAT/ LIST(100)
DATA A /3.5/, LIST /100*5/
END
```

Procedures

Procedure subprograms include both function subprograms and subroutine subprograms. Both user-written subprograms and the intrinsic subprograms supplied as part of the CONVEX Fortran system are supported by CONVEX Fortran.

Procedure subprograms can contain executable statements. In general, after its initial defining statement, a procedure subprogram can contain any statement except a PROGRAM, BLOCK DATA, SUBROUTINE, or FUNCTION statement.

Dummy and actual arguments

Dummy arguments are classified as variables, arrays, or dummy procedures. They are used in statement functions, function subprograms, and subroutine subprograms to indicate the number and types of actual arguments to be transferred. 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 specify the entities that are to be associated with the dummy arguments. They may be constants, symbolic names of constants, function references, expressions, arrays and array elements, character substrings, alternate return specifiers, or subprogram names. 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 also must 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 statement:

```
SUBROUTINE SAMPLE (R,L)
```

specifies R and L as dummy arguments. When the subroutine

```
CALL SAMPLE (B,80)
```

executes, the actual arguments (B, 80) replace the dummy arguments (R, L). 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.

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 can 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.

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 must not be passed as actual arguments if the called routine, or any routine that it calls, accesses that element from the COMMON block.

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.

Assumed-size arrays

An asterisk specifies 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 or an implied DO loop without subscripts
- 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 is a noncharacter array name. When the actual 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 one of the following:

- A character array name
- Character array element name
- Character array element substring

and:

- *s* is the character storage unit of an array where the actual argument begins.
- *n* is the number or character storage units in this array.
- *l* is the length of an element of the dummy array.

If an assumed size dummy array has *n* dimensions, the product of the sizes of the first *n-1* dimensions must not exceed the size of the array.

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 of 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 (for example, 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 (for example, "SAM").

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 dummy 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 are 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 specify the dummy argument length as an assumed-length character argument (for example, `*(*)`), the length of the associated actual argument is used.

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. When a dummy argument of type CHARACTER is an array name, the restriction on length is for the entire array and not for each array element.

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 character argument length specifications.

```
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 is defined as

```
SUBPROGRAM MYEXAM (A)
CHARACTER A*(*)
     .
     .
     .
END
```

all three of the CALL statements are valid. The length of the dummy argument A is determined by the length of the corresponding actual argument.

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, or
- 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 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 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.

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.

Example:

```
SUBROUTINE EXAM(D3,*,E2,*)
```

The alternate return argument 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 following form:

**label* or *&label*

Functions

A function can be an intrinsic function, a statement function, or an external function (function subprogram), all of which 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 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 an optional list of actual arguments separated by commas. If you do not include arguments, you must still include the enclosing parentheses.

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 for intrinsic functions.

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 type `REAL`, `DOUBLE PRECISION`, *`REAL*16`*, 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 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 following form:

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

where *inf* is an intrinsic name and *a* is the argument on which the function operates.

Built-in functions

Built-in functions allow a Fortran program to pass arguments to a program that is not written in Fortran.

%REF and %VAL functions

Two built-in functions—%REF and %VAL—can be used in the argument list of a CALL statement or function reference to change the form of the argument. Such a change can be necessary if you must call subprograms written in languages other than Fortran, because the actual argument may have to be passed in a form different from that used by Fortran. These two functions specify how to pass the argument 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.

There are two built-in argument list functions:

- *%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.*

a is an actual argument.

Table 28 shows the Fortran argument-passing defaults and allowable uses of %REF and %VAL.

Table 28 Built-in functions and defaults for argument lists

Data type	Default	Functions allowed	
		%REF	%VAL
LOGICAL (*1,2,4)	REF	Yes	Yes†
LOGICAL*8	REF	Yes	No
INTEGER (*1,2,4)	REF	Yes	Yes†
INTEGER*8	REF	Yes	No
REAL*4	REF	Yes	Yes
REAL*8	REF	Yes	No
REAL*16	REF	Yes	No
COMPLEX	REF	Yes	No
CHARACTER	REF	Yes	No
Hollerith	REF	No	No
Array name			
Numeric	REF	Yes	No
Character	REF	Yes	No
Procedure name			
Numeric	REF	Yes	No
Character	REF	Yes	No

†If a logical or integer value occupies less than 32 bits of storage, it is converted to a 32-bit value by sign extension.

%LOC function

The %LOC built-in function computes the internal address of a storage element, as in the following example:

```
IADDR = %LOC(v)
```

*where v is a variable name, array element name, array name, character substring name, or external procedure name, and IADDR is an INTEGER*4 variable.*

*The %LOC built-in function produces an INTEGER*4 value that represents the location of its argument. Addresses from user programs are BYTE addresses.*

The use of %LOC is limited to arithmetic expressions.

Statement functions

The statement function is a nonexecutable, user-defined, single-statement procedure. 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 defined 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 in the function definition must be unique 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 expression). 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, any expressions can be used as actual arguments except array names and character expressions involving concatenation of an operand whose length specification is an asterisk (*) in parentheses, unless the operand is the symbolic name of a constant.

Function subprograms

A function subprogram is a separate program unit that includes a FUNCTION statement followed by a series of statements that define the computing procedure. The calling program unit

references a function; the function's statements execute, and through a RETURN or END statement, a single scalar or array value returns to the function reference in the calling unit.

You must begin a function subprogram with the FUNCTION statement. You can include any statements except a BLOCK DATA, SUBROUTINE, PROGRAM, or another FUNCTION statement within a 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*. You can include ENTRY statements to provide multiple entry points to a subprogram.

A function specified in a subprogram can reference other subprograms but cannot reference itself, directly or indirectly. It must assign a value to its symbolic name at least once.

Logical and numeric functions
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. This specifies the return type for the function. Additional statements, described later in this chapter, are required to establish an array-valued return value.

nam

is the symbolic name of the function subprogram. If you neither specify *typ* nor declare the *nam* in a later type statement, this 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.

Array-valued functions

CONVEX Fortran supports user-written functions that return array values. For a function to return an array, the following must take place:

- The function must be declared external in an EXTERNAL specification statement within the calling program unit. This indicates that the function's symbolic name refers to a subprogram rather than a variable.
- The function's symbolic name must appear in a type statement in the calling program unit to properly type the function.
- Within the function, the array being returned must have its shape established and must have a value assigned for each element.

The following example illustrates how to declare and call an array-valued function:

```
PROGRAM MAIN
EXTERNAL ARRFUN             !Declare the function
                            !to be external.
INTEGER A(10),ARRFUN(10)    !Establish the type.
...
A = ARRFUN(32)              !Call the function.
PRINT *, A
END

INTEGER FUNCTION ARRFUN(VAL)
DIMENSION ARRFUN(10)        !Establish the shape.
INTEGER VAL
DO I = 1, 10                !Assign each element.
   ARRFUN(I) = INT ((VAL / I) * 5 )
END DO
END
```

Character function

In character functions, the CHARACTER FUNCTION statement, *with the CONVEX extension of* *n and optional parentheses, has the form:

```
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*.

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.

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; 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 which 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, 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 end it with an END statement. It specifies the name of the subroutine and the arguments used by the subroutine. The SUBROUTINE statement has the following form:

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

where

name

is the symbolic name of the subroutine. Because the subroutine has no data type, you need not apply the naming rules. Because 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, if the subroutine uses alternate returns, an asterisk (*). Separate the argument list items with 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 can be used to specify multiple entry points for subroutines.

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 association between actual dummy arguments is not retained between references.

An entry name or the name of the function subprogram defines all associated names of the same data type. 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 is executed 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.

RETURN statement

The RETURN statement is used to return from a subroutine subprogram to one of several alternate 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 following form:

```
RETURN [e]
```

where *e*—which may be specified only in subroutine subprograms—is an optional integer expression that specifies an alternate statement in the calling program that 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 or ampersand 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 (for example, *30 or &30).

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
```

In the preceding example, 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.

Use of 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 ends the association between the dummy arguments and the current actual arguments. If the `-re` compiler option is specified, all local data that did not appear in a SAVE statement becomes undefined.

When used within 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.

SPP Series synchronization 12

On SPP Series computers, CONVEX Fortran supports commonly used synchronization features with library routines, compiler directives, and intrinsic routines. The directives and routines described in this chapter are useful only in programs compiled at optimization level `-O3`.

For additional information about SPP Series programming, refer to the *Exemplar Programming Guide*.

Note

The features described in this chapter are available only on CONVEX SPP Series computers.

Two data types are provided for use with these features, the `BARRIER` and `GATE` data types. Barrier routines are used for synchronizing multiple threads in a program. Gates provide a mechanism for restricting execution of a block of code to a single thread at a time. In addition, the library routines described in the "Process information intrinsic routines" section return information about the hypernodes, processors, threads, and parallelism of a process, which can be used in synchronizing a program's events.

The compiler directives in Table 29 are used to declare `GATE` and `BARRIER` variables and establish synchronized regions of code.

Table 29 Synchronization compiler directives (SPP Series only)

Compiler directive	Data type accepted	Operation
`BARRIER`	`BARRIER`	Declare a variable of type `BARRIER`.
`GATE`	`GATE`	Declare a variable of type `GATE`.
`ORDERED_SECTION`	`GATE`	Force all threads to execute a specified block of code individually and in thread order.
`END_ORDERED_SECTION`	none	Mark the end of an ordered section block.

Table 29 (continued) Synchronization compiler directives (SPP Series only)

Compiler directive	Data type accepted	Operation
CRITICAL_SECTION	GATE	Restrict execution of a specified block of code to one thread at a time.
END_CRITICAL_SECTION	none	Mark the end of a critical section block.

Table 30 lists the routines available for use with the GATE and BARRIER data types. All functions in Table 30 return zero (0) to indicate successful completion and return negative one (-1) to indicate failure. Note that these return values do *not* correspond to boolean .TRUE. and .FALSE. values. Functions in this table whose names end with "_8" return INTEGER*8 values, and all others return INTEGER values.

Table 30 Synchronization library routines (SPP Series only)

Library routine	Data type accepted	Operation
ALLOC_GATE ALLOC_GATE_8	GATE	Allocate storage for a specified gate.
FREE_GATE FREE_GATE_8	GATE	Release storage assigned to a gate.
LOCK_GATE LOCK_GATE_8	GATE	Set the state of the specified gate to locked.
UNLOCK_GATE UNLOCK_GATE_8	GATE	Set the state of the specified gate to unlocked.
COND_LOCK_GATE COND_LOCK_GATE_8	GATE	Set the state of the specified gate to locked only if the gate is immediately available.
ALLOC_BARRIER ALLOC_BARRIER_8	BARRIER	Allocate space for the specified barrier.
FREE_BARRIER FREE_BARRIER_8	BARRIER	Release storage assigned to a barrier.
WAIT_BARRIER WAIT_BARRIER_8	BARRIER	Mark a point at which threads synchronize.

All library routines and compiler directives listed in Table 29 and Table 30 are explained in further detail in the sections that follow.

Barriers

The BARRIER data type and its supporting directive and routines enable programmers to synchronize a specified number of threads within a program.

BARRIER variables may appear only in COMMON statements, in DIMENSION statements, and as dummy and actual arguments. They may not be initialized by DATA or assignment statements.

Declaring barriers

Variables of data type BARRIER must be both declared and allocated before they can be used. The BARRIER compiler directive declares a variable of type BARRIER and may appear only where type declarations are legal. The BARRIER directive overrides any previous type declarations. It is an error for variables to appear in subsequent type statements after being declared type BARRIER.

This directive has the form

```
C$DIR BARRIER(barr-name [, barr-name...])
```

where *barr-name* is the name of the variable to be declared. Multiple barriers may be declared at once. The following code declares a barrier named "my_barrier":

```
C$DIR BARRIER(my_barrier)
```

Arrays of barriers can be created using the DIMENSION statement. The following statements would declare a one-dimensional BARRIER array with five elements, numbered 2 to 6.

```
C$DIR BARRIER(bar_arr)
      DIMENSION bar_arr(2:6)
```

Allocating barriers

Each BARRIER variable must be allocated through a call to the ALLOC_BARRIER or the ALLOC_BARRIER_8 routine before it

can be used. The ALLOC_BARRIER routine allocates space for a barrier structure and has the form

intvar = ALLOC_BARRIER(*barr-var*)

where

intvar is a variable of type INTEGER that receives the result value of ALLOC_BARRIER

and

barr-var is a BARRIER variable to be allocated that has been declared using the BARRIER compiler directive

The variable *intvar* is assigned a result value of zero (0) if ALLOC_BARRIER successfully allocates storage for the barrier *barr-var*. Otherwise *intvar* is assigned a result value of negative one (-1).

ALLOC_BARRIER_8 performs the same function as ALLOC_BARRIER, but returns a result value of type INTEGER*8 (instead of type INTEGER).

Deallocating (freeing) barriers

To deallocate a barrier, use the FREE_BARRIER or the FREE_BARRIER_8 library routine. Both functions release the space assigned to the barrier and make the barrier available to be reallocated. A barrier cannot be allocated more than once without an intervening call to FREE_BARRIER or FREE_BARRIER_8.

The FREE_BARRIER routine has the form

intvar = FREE_BARRIER(*barr-var*)

where

intvar is a variable of type INTEGER that is assigned FREE_BARRIER's result value

and

barr-var is a BARRIER variable to be deallocated (freed) that has been declared using the BARRIER compiler directive

The variable *intvar* is assigned a result value of zero (0) if FREE_BARRIER successfully deallocates storage for the barrier *barr-var*. Otherwise *intvar* is assigned a result value of negative one (-1).

FREE_BARRIER_8 performs the same function as FREE_BARRIER, but returns a result value of type INTEGER*8 (instead of type INTEGER).

Following deallocation, a BARRIER variable cannot be used unless it is allocated again with a call to ALLOC_BARRIER or ALLOC_BARRIER_8.

Synchronizing threads using barriers

The WAIT_BARRIER and WAIT_BARRIER_8 routines enable programmers to specify a point in a program at which a specified number of threads will synchronize.

The WAIT_BARRIER routine has the following form

```
intvar = WAIT_BARRIER(barr-var, num)
```

where

intvar is a variable of type INTEGER that receives the result value of WAIT_BARRIER

barr-var is a BARRIER variable that has been declared using the BARRIER compiler directive

and

num is the number of threads WAIT_BARRIER expects to arrive before it simultaneously releases them

The variable *intvar* is assigned a result value of zero (0) if WAIT_BARRIER completes successfully. Otherwise *intvar* is assigned a result value of negative one (-1).

WAIT_BARRIER_8 performs the same function as WAIT_BARRIER, but returns a result value of type INTEGER*8 (instead of type INTEGER).

The call in the following example specifies that three threads of barrier my_barrier must reach this point in the program.

```
X = WAIT_BARRIER(my_barrier, 3)
```

Each of the threads waits at the specified point in the program until the third thread reaches that point, then all three threads are released simultaneously.

Gates

Several CONVEX Fortran subroutines provide ways to limit execution of a block of code to one thread at a time. By using the

GATE data type and its attendant library routines, you can lock and unlock gates to prevent or permit access to portions of a program.

GATE variables may appear only in COMMON statements, in DIMENSION statements, and as dummy and actual arguments. They may not be initialized by DATA or assignment statements.

Additional methods of restricting the execution of a block of code appear in the "Critical sections" portion of this chapter.

Declaring gates

Variables of type GATE must be declared using the GATE compiler directive, which may appear only where type declarations are legal. The GATE directive overrides any previous type declarations. It is an error for variables to appear in subsequent type statements after being declared type GATE.

The GATE directive has the form

```
C$DIR GATE(gate-name [, gate-name...])
```

where *gate-name* is the name of the variable to be declared. Multiple gates may be declared at once.

The following line declares a GATE variable named "my_gate".

```
C$DIR GATE(my_gate)
```

Arrays of gates can be created using the DIMENSION statement. The following statements would declare a one-dimensional GATE array with ten elements:

```
C$DIR GATE(gate_arr)
      DIMENSION gate_arr(10)
```

Allocating gates

Like barriers, gates also must be allocated before they can be used. The ALLOC_GATE and ALLOC_GATE_8 library functions allocate space for a specified gate and set the gate's state to unlocked. The ALLOC_GATE function has the following form:

```
intvar = ALLOC_GATE(gate-var)
```

where

intvar is a variable of type INTEGER that receives the result value of ALLOC_GATE

and

gate-var is a GATE variable to be allocated that has been declared using the GATE compiler directive

The variable *intvar* is assigned a result value of zero (0) if ALLOC_GATE successfully allocates (initializes) storage for the gate *gate-var*. Otherwise *intvar* is assigned a result value of negative one (-1). Note that gates are set to an unlocked state upon allocation.

ALLOC_GATE_8 performs the same function as ALLOC_GATE, but returns a result value of type INTEGER*8 (instead of type INTEGER).

Only deallocated gates may be allocated. If a gate is allocated more than one place in a program, the gate must be deallocated between calls to ALLOC_GATE or ALLOC_GATE_8 by using the FREE_GATE or FREE_GATE_8 routine.

Deallocating (freeing) gates

FREE_GATE and FREE_GATE_8 release space assigned to a gate, leaving the GATE variable declared but uninitialized. The FREE_GATE library function has the following form:

intvar = FREE_GATE(*gate-var*)

where

intvar is a variable of type INTEGER that is assigned FREE_GATE's result value

and

gate-var is a GATE variable to be deallocated (freed) that has been declared using the GATE compiler directive

The variable *intvar* is assigned a result value of zero (0) if FREE_GATE successfully deallocates storage for the gate *gate-var*. Otherwise *intvar* is assigned a result value of negative one (-1).

FREE_GATE_8 performs the same function as FREE_GATE, but returns a result value of type INTEGER*8 (instead of type INTEGER).

Following deallocation, a GATE variable cannot be used unless it is allocated again with a call to ALLOC_GATE or ALLOC_GATE_8.

Coordinating execution using gates

By using CONVEX Fortran's routines for manipulating GATE variables, programmers can establish points within a program where access to blocks of code is restricted based on the state of a GATE variable.

Locking and unlocking gates

The LOCK_GATE (or LOCK_GATE_8) and UNLOCK_GATE (or UNLOCK_GATE_8) library routines control the state of GATE variables by setting them to either a locked or unlocked state.

A call to LOCK_GATE or LOCK_GATE_8 sets the specified gate variable to a locked state. The LOCK_GATE function uses the format described here:

intvar = LOCK_GATE(*gate-var*)

where

intvar is a variable of type INTEGER that is assigned LOCK_GATE's result value

and

gate-var is a GATE variable to be locked that has been declared using the GATE compiler directive

The variable *intvar* is assigned a result value of zero (0) if LOCK_GATE successfully locks the gate *gate-var*. Otherwise *intvar* is assigned a result value of negative one (-1). If *gate-var* cannot be immediately acquired, the calling thread waits until *gate-var* is available for locking.

LOCK_GATE_8 performs the same function as LOCK_GATE, but returns a result value of type INTEGER*8 (instead of type INTEGER).

The UNLOCK_GATE and UNLOCK_GATE_8 functions set a gate variable to an unlocked state. UNLOCK_GATE has the following form:

intvar = UNLOCK_GATE(*gate*)

where

intvar is an INTEGER variable that is assigned a return value of either 0 (if the gate is successfully unlocked) or -1 (if the function cannot unlock the gate), and *gate* is the GATE variable to be unlocked.

UNLOCK_GATE_8 performs the same function as UNLOCK_GATE, but returns a result value of type INTEGER*8 (instead of type INTEGER).

The LOCK_GATE and UNLOCK_GATE routines used in the following example provide each thread exclusive access to the enclosed block of code.

```
      INTEGER X
C$DIR GATE(my_gate)
      .
      .
      .
C$DIR LOOP_PARALLEL
      DO I = 1, N
      .
      .
      .
      X = LOCK_GATE(my_gate)
      TOT = TOT + A(I)
      X = UNLOCK_GATE(my_gate)
      .
      .
      .
      ENDDO
```

When each thread calls the LOCK_GATE routine, the routine checks the state of the GATE variable my_gate. If my_gate is locked, the calling thread waits for it to be unlocked before proceeding. The LOCK_GATE function returns an INTEGER value, which is assigned to X and is ignored in this example.

The call to UNLOCK_GATE_8 returns an INTEGER*8 value, which is assigned to Y and also is ignored here. The variable Y is used only to receive the assignment from UNLOCK_GATE_8. UNLOCK_GATE_8 releases my_gate from exclusive access by setting it to an unlocked state. Once this is done, another thread waiting for my_gate to be unlocked (if one is waiting) is permitted by LOCK_GATE to continue executing and LOCK_GATE locks the variable my_gate, thus ensuring the thread exclusive access to the restricted block of code.

Conditionally locking gates

An alternative to the LOCK_GATE library routine is COND_LOCK_GATE. COND_LOCK_GATE and COND_LOCK_GATE_8 differ from LOCK_GATE and LOCK_GATE_8 in that they do not wait for exclusive access to a gate if it is not immediately granted.

COND_LOCK_GATE and COND_LOCK_GATE_8 return zero (0) if the gate was acquired or negative one (-1) if the gate could not be acquired.

The COND_LOCK_GATE routine has the form

```
intvar = COND_LOCK_GATE(gate)
```

where

intvar is an INTEGER variable that is assigned the return value of either zero (0) if the gate was acquired, or negative one (-1) if the gate could not be acquired

and

gate is the GATE variable to be locked.

The COND_LOCK_GATE routine acquires a gate for exclusive access if it can do so without waiting. If the specified gate already is locked, COND_LOCK_GATE does not wait for the gate to be released and does not receive the lock.

COND_LOCK_GATE_8 performs the same function as COND_LOCK_GATE, but returns a result value of type INTEGER*8 (instead of type INTEGER).

Critical sections

Another mechanism for restricting the execution of a block of code to one thread at a time is the CRITICAL_SECTION and END_CRITICAL_SECTION pair of directives. These directives together function as a LOCK_GATE – UNLOCK_GATE pair.

Note

Although all threads will execute the code contained between CRITICAL_SECTION and END_CRITICAL_SECTION directives, the order in which they execute the code is arbitrary; do not use critical sections to isolate dependencies

The directives denoting a critical section must appear in the same control flow within the same procedure, but they do not have to be in the same procedure as the parallel constructs in which they are used. For example, a loop running in parallel may call a subprogram that contains these directives.

The CRITICAL_SECTION directive has the following form

```
C$DIR CRITICAL_SECTION[(gate)]
```

where *gate* specifies an optional GATE variable to be used for restricting access to the critical section of code.

The GATE variable can be used to synchronize among multiple tasks and to distinguish among different critical sections. If a

gate is specified, it must be unlocked prior to its use and must be allocated outside of any parallel construct in which it is used.

The END_CRITICAL_SECTION compiler directive has the form

```
C$DIR END_CRITICAL_SECTION
```

If a GATE variable is specified in a CRITICAL_SECTION directive, the subsequent END_CRITICAL_SECTION directive unlocks the GATE variable.

In the following example, the directives marking the critical section ensure that the SUM operation will be performed by only one thread at a time.

```
C$DIR LOOP_PARALLEL
      DO I = 1, N
         ...
C$DIR CRITICAL_SECTION
         SUM = SUM + X(I)*Y(I)
C$DIR END_CRITICAL_SECTION
         ...
      END DO
```

Each thread waits for exclusive access at the CRITICAL_SECTION directive and relinquishes access at the END_CRITICAL_SECTION directive. In this simple case, no gate variable is specified with the CRITICAL_SECTION directive because none is needed.

Ordered sections

The ORDERED_SECTION and END_ORDERED_SECTION compiler directives force all threads to execute a designated section of code one thread at a time in thread order (the order in which the threads are spawned). These directives are useful only in LOOP_PARALLEL(ORDERED) loops.

The ORDERED_SECTION directive is similar to CRITICAL_SECTION in that both limit execution of a code block to one thread at a time. However, unlike CRITICAL_SECTION, ORDERED_SECTION also ensures that the code executes in thread order.

The ORDERED_SECTION directive has the following form:

```
C$DIR ORDERED_SECTION (gate)
```

where *gate* specifies a gate to be used to control entry into the ordered section.

The END_ORDERED_SECTION directive specifies the point where the ordered section ends; this directive does not accept the gate parameter.

In the following example, the DO loop is parallelizable except for the backward loop-carried dependency (LCD) in the ordered section, where each assignment to X(I) depends on the value of X(I-1). The ordered section directives isolate the LCD and, in combination with the LOOP_PARALLEL(ORDERED) directive, forces the enclosed code to execute in loop order by one thread at a time, thus enabling the compiler to parallelize the loop.

```
C$DIR GATE(my_gate)
      X = ALLOC_GATE(my_gate)
      ...
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, 10
         ...
C$DIR ORDERED_SECTION(my_gate)
         X(I) = X(I-1) + A(I)
C$DIR END_ORDERED_SECTION
         ...
      END DO
```

When this loop executes, the ordered section first is evaluated for I=2, then for I=3, I=4, and so on. This guarantees that when X(8) is assigned, the value of X(7) already will have been assigned and thus can be used to compute the value for X(8).

Process information intrinsic routines

CONVEX Fortran supports several intrinsic routines that provide information about processes running on CONVEX SPP Series machines. These routines, when used in a Fortran program, return information about the program's current state of execution.

Each of the routines described in this section has a version that returns INTEGER values and one that returns INTEGER*8 values. Versions of these routines also are provided for the CONVEX C language on SPP Series machines.

Table 31 Process information intrinsic routines (SPP Series only)

Library routine	Return type	Description
NUM_PROCS() NUM_PROCS_8()	INTEGER INTEGER*8	Returns the number of processors the process is using at runtime.

Table 31 (continued) Process information intrinsic routines (SPP Series only)

Library routine	Return type	Description
`NUM_THREADS()` `NUM_THREADS_8()`	`INTEGER` `INTEGER*8`	Returns the number of threads the process is using at runtime.
`NUM_NODES()` `NUM_NODES_8()`	`INTEGER` `INTEGER*8`	Returns the number of nodes on which the process is running.
`MY_THREAD()` `MY_THREAD_8()`	`INTEGER` `INTEGER*8`	Returns the spawn thread id.
`MY_NODE()` `MY_NODE_8()`	`INTEGER` `INTEGER*8`	Returns the node id of the calling thread.
`NUM_NODE_THREADS()` `NUM_NODE_THREADS_8()`	`INTEGER` `INTEGER*8`	Returns the number of threads that are on the node the calling thread is on.
`LEVEL_OF_PARALLELISM()` `LEVEL_OF_PARALLELISM_8()`	`INTEGER` `INTEGER*8`	Returns the level of parallelism the current process is in. See Table 32 for return values.
`MEMORY_TYPE_OF_STACK()` `MEMORY_TYPE_OF_STACK_8()`	`INTEGER` `INTEGER*8`	Returns the memory class the current thread stack is allocated from. See Table 33 for values.

Calls to these intrinsic functions may appear in executable statements anywhere an integer expression is valid.

A call to the `LEVEL_OF_PARALLELISM` intrinsic routine returns one of values in Table 32 to describe the calling process's current parallel status:

Table 32 `LEVEL_OF_PARALLELISM` return values (SPP Series only)

Numeric value	Current parallel status
0	not parallel
1	asymmetric thread is active
2	node-way parallelism
4	thread-way parallelism on a node
8	thread-way parallelism
16	asymmetric thread-way parallelism

A call to the MEMORY_TYPE_OF_STACK intrinsic routine returns a value that indicates the memory class from which the current thread stack is allocated. These return values are listed in Table 33.

Table 33 MEMORY_TYPE_OF_STACK return values (SPP Series only)

Return value	Description
0	far-shared memory
1	near-shared memory
2	node-private memory

For detailed information about writing optimal parallel code on CONVEX SPP Series computers, consult the *Exemplar Programming Guide*.

SPP Series optimizations 13

This chapter discusses the various optimization options available for use with the SPP Series Fortran compiler. Also consult the *Exemplar Programming Guide* for detailed explanations of the optimizations performed at each optimization level. For information about optimally compiling Fortran code for execution on C Series machines, refer to the *Fortran Optimization Guide.*

Optimization options

Five optimization options are available for use with the CONVEX Fortran compiler. These options are specified on the compiler command line along with any other compiler options you wish to use. These optimization levels are summarized in Table 34.

Table 34 Optimization options

Option	Description
`-no`	Machine instruction level scalar optimizations. This option is the default.
`-O0`	Basic block level scalar optimizations.
`-O1`	Program unit level scalar optimizations and global register allocation.
`-O2`	Global instruction scheduling and data localization optimizations.
`-O3`	Parallel optimizations.

These options are cumulative; each option retains the optimizations of the previous option. For example, entering the following command line

```
fc -O2 foo.f
```

would compile the Fortran program `foo.f` with all `-O2`, `-O1` and `-O0` level optimizations shown in Table 34.

-no level optimizations

At optimization level `-no`, the compiler performs scalar optimizations that span no more than a single source statement. These optimizations create object code that fully uses the scalar features of the SPP Series PA-RISC architecture.

Instruction scheduling

Instruction scheduling rearranges machine instructions to use the computer's functional units most effectively. Each CPU on an SPP Series system has multiple functional units on which operations execute simultaneously; this concurrent execution within a single processor is distinct from parallel processing, which occurs on multiple processors at optimization level `-O3`.

Span-dependent instructions

There are several ways to specify a branch in the PA-RISC machine language; short branches can be more efficiently executed than long branches. The SPP Series compilers automatically generate branches using the most efficient and appropriate branching techniques.

Register allocation

The SPP Series compilers use a technique for allocating registers that fully exploits the PA-RISC register set. This allows grouping of register loads and concurrent execution of instructions, and reduces register conflicts.

Tree-height reduction

The compiler represents expressions internally as trees. When they involve floating point numbers, these trees are optimized by *tree-height reduction* or *balancing*.

The deeper a tree representing an expression is, the more time required to evaluate the expression. If a particular evaluation order is not specified with parentheses, the compilers choose one that minimizes the depth of the expression and maximizes instruction pipelining, while maintaining the arithmetically correct evaluation order.

Short-circuit evaluation of conditionals in Fortran

Short circuiting increases the efficiency of `IF` statements by skipping irrelevant tests when logical operators are involved in the conditional. CONVEX Fortran short-circuits evaluation of `IF` statements that contain `.AND.` and `.OR.` operators that have logical operands and are used in a logical context. Take, for example, the following `IF` statement:

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if `(A .EQ. B)` evaluates to false, the evaluation of `F(G)` and the `THEN` portion of the statement is skipped.

The compiler short-circuits the evaluation of conditionals by default. You can disable short-circuiting in Fortran by specifying the `-nosc` flag on the compiler command line.

-O0 Level optimizations

At optimization level `-O0`, the compiler performs scalar optimizations within a basic block. A basic block is a sequence of statements with a single entry point and a single exit. At level `-O0` the compiler also continues to perform the optimizations performed at `-no`.

Instruction scheduling

At optimization level `-O0` and above, instructions from *multiple* statements, as well as those from single statements, are scheduled as a group.

Redundant-assignment elimination

Redundant-assignment elimination removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated.

Assignment substitution

Assignment substitution eliminates redundant loads. The compiler stores the value assigned to a variable in a register and replaces subsequent references to that variable with the register value. Because the compiler substitutes assignments, you rarely need to optimize a program by replacing a variable reference with a constant value in the source code.

Common-subexpression elimination

The compiler recognizes subexpressions (portions of expressions) that repeat within a basic block. The compiler retains the value of the subexpression in a register, which eliminates redundant computations and register loads. For example, the compiler recognizes `B+C` as a common subexpression of `A+B+C+D` and `B+E+C`, and calculates the subexpression only once. The compiler also eliminates redundant array address calculations.

Redundant-use elimination

This optimization is a special case of common subexpression elimination where the subexpression is a variable. The compiler detects multiple references to a variable between assignments and retains the value of the variable in a register. This action helps eliminate redundant register loads.

Constant propagation and folding

After assigning a constant to a variable, the compiler replaces subsequent references to the variable with the constant. For example, if you write `X=5`, the compiler replaces `X` with 5 within that basic block or until a new value is assigned to the variable. This is known as *constant propagation*, which is a form of assignment substitution.

The compiler also replaces operations on constants with the result of the operation. This is known as *constant folding*. For example, it replaces `Y=5+7` with `Y=12`. It then propagates the constant value to replace future references to `Y` within the basic block. The Fortran compiler also propagates and folds values assigned to names in `PARAMETER` statements.

The compiler folds the most frequently used Fortran intrinsics when they are applied to constant arguments. For example,

`SIN(0.0)` becomes `0.0`. The compiler also folds exponentiation involving constants. For example, `3**3` becomes `27`.

The compiler type-converts constants, if necessary, before propagating and folding them. If a Fortran program contains the expression `X=1`, where `X` is `REAL`, the compiler converts `1` to `1.0` before propagating it.

If an integer overflow occurs as a result of constant folding in Fortran, the compiler reports "Integer constant truncation." If a floating-point overflow occurs, the compiler reports "Real constant either too large or too small." Floating-point under-flow always results in zero. If any of these messages or conditions occur, eliminate the offending operation or bring the value of the constant within acceptable bounds.

Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown in the following Fortran examples.

Original expression	Optimized expression
`X + 0`	`X`
`X * 1`	`X`
`X * 0`	`0`
`K .AND. -1`	`K`
`K .AND. 0`	`0`
`K .OR. -1`	`-1`
`K .OR. 0`	`K`
`-1 * X`	`-X`
`X - X`	`0`
`X / -1`	`-X`
`(-1) ** K`	`1-((K .AND. 1) * 2)`
`X ** 0.5`	`SQRT(X)`
`X ** 0`	`1`
`1 ** X`	`1`
`X / X`	`1`
`0 - X`	`-X`
`0 / X`	`0`
`SIN(X) * COS(X)`	`0.5 * SIN(2X)`
`SIN(X) / COS(X)`	`TAN(X)`

The compiler performs obvious variations of these operations for the commutative operators. For example, in Fortran the compiler converts `X+(0+Y)` to `X+Y`.

-O1 Level optimizations

At optimization level -O1, global optimization is done across a group of basic blocks but within a single procedure. The -O1 option performs global, basic-block, and machine instruction level optimizations.

Constant propagation and folding

Propagating and folding constants at the procedure level is analogous to performing the same operations at the basic-block level. The scope of the optimization is now an entire procedure.

The compiler propagates and folds constants globally at optimization level -O1 and higher, which eliminates the need to propagate constants by hand in programs compiled at these levels.

Redundant-assignment elimination

At optimization level -O1, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. The Fortran compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and COMMON variables.

If the right side of a redundant assignment statement contains a procedure call, the compiler eliminates the assignment and retains the call. If a function appearing in an assignment has no side effects, the compiler can eliminate the function call, as well as the assignment, thus improving efficiency. A procedure has no side effects if it:

- doesn't modify the value of an argument
- doesn't modify the value of a COMMON variable
- doesn't modify the value of local static variables used on subsequent calls
- doesn't perform input or output
- doesn't call another procedure that has side effects.

Existing procedure compilers cannot automatically determine whether a side effect exists. The CONVEX Fortran compiler

eliminates procedure calls only if you explicitly request it with the NO_SIDE_EFFECTS directive. The form of this directive is:

```
C$DIR NO_SIDE_EFFECTS (func_list)
```

where *func_list* is a list of procedure names separated by commas. The directive must precede the call to the procedure that does not contain side effects. For more information about the NO_SIDE_EFFECTS directive, see *Fortran User's Guide* Appendix A, "Compiler directives."

Dead-code elimination

If, as a result of constant propagation and folding, the compiler can fold an arithmetic or logical expression in a Fortran IF statement to .TRUE. or .FALSE., the compiler eliminates any unreachable code that results.

Copy propagation

The compiler can replace a variable with another variable to which it has been equated. This is called *copy propagation*. For example, after evaluating the statement X=Y, the compiler replaces later occurrences of X with Y.

Common subexpression elimination

The compiler eliminates common subexpressions at the global level. The compiler retains the value of the common subexpression in a register if one is available; otherwise, the compiler assigns the value to a temporary variable. The compiler then replaces subsequent occurrences of the common subexpression with references to the register or temporary variable.

Code motion

Code motion is the movement of invariant expressions out of loops. An invariant expression yields the same result on every iteration of a loop.

In the following example, all variables used in the assignment to A remain invariant within the loop. The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

```
SUBROUTINE GCM
REAL AR(10)
...
DO I = 1, 10
  A = C / (-(E * B) + SQRT(C))
  AR(I) = A + B * C
ENDDO
...
END
```

If an invariant expression does not lie on a path to all loop exits (for instance, when it appears in an `IF` statement), the compiler does not move the invariant expression unless you use the `-uo` (potentially unsafe optimizations) compiler option.

Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation (possibly nonarithmetic) that executes more quickly. Such replacements are called strength reductions.

Explicit arithmetic reductions

The compiler can reduce the strength of various arithmetic operations. For example, the compiler transforms integer multiplication on positive numbers by 2, 4, 8, and 16 into integer shifts, as shown in the following Fortran code:

`J * 2` becomes `IISHFT(J, 1)`
`J * 4` becomes `IISHFT(J, 2)`

Multiplication involving integer constants is reduced to addition:

`X * 3` becomes `X + X + X`

When the `-uo` (potentially unsafe optimizations) command line option is specified, division by a constant is reduced to multiplication:

`X/C` becomes `D*X` where `D=1/C`

Because `C` is a constant, `D` also is a constant, which can be computed at compile time.

Induction variables and constants

The compiler can reduce the strength of operations to optimize loop induction variables and loop constants. Multiplications within a loop that calculate the address of a subscripted variable are often candidates for strength reduction.

The compiler does not reduce operations that only involve REAL variables. Because floating-point arithmetic is imprecise, reduced operations do not always yield equivalent results. If an expression does not lie on a path to all loop exits (for instance, when it appears in an IF statement), the compiler does not reduce the expression unless you use the -uo option.

Global register allocation

Global register allocation (GRA) attempts to store commonly-referenced scalar variables in registers throughout the code where they are most frequently accessed, thus reducing the need for slower main memory accesses. The compiler automatically determines which scalar variables are the best candidates for GRA and allocates registers accordingly. GRA is enabled by default at optimization levels -O1 and above.

GRA can sometimes cause wrong answers in code that violates the ANSI standard argument-passing conventions. This problem occurs when a constant is passed into a subroutine that potentially attempts an assignment to it. If the constant is allocated a register, even if the code does not make an assignment to the constant, an error will occur when the register is stored back to the constant. To avoid this situation, specify the -nga compiler option, which disables GRA for arguments passed by reference.

GRA also can sometimes cause problems when parallel threads attempt to update a shared variable that has been allocated a register. In this case, each thread allocates a register for the shared variable, making it unlikely that the copy in main memory will be correctly updated as each thread executes. To disable GRA for shared variables that are visible to multiple threads, use the -ngs compiler option.

-O2 Level optimizations

On SPP Series machines, the primary goal of -O2 optimizations is *data localization*. Data localization keeps heavily used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

Loops that manipulate arrays are the main candidates for localization optimizations. Most of these loops are eligible for

the various transformations the compiler performs at level -O2 to achieve localization.

In addition to localization, the compiler performs global instruction scheduling and all basic block and machine-instruction level optimizations.

Complicated loops, such as a matrix multiply algorithm, and loops that manipulate large arrays often can be transformed by the compiler such that far more data reuse is possible, resulting in greater performance.

The following sections explain the loop transformations that aid data localization. For more information about -O2 level optimizations on SPP Series machines, or information about conditions that prevent -O2 optimizations, consult the *Exemplar Programming Guide.*

Strip mining

Strip mining is a fundamental -O2 transformation that is used by loop blocking and, in a sense, by parallelization.

Strip mining involves splitting a single loop into a nested loop; the resulting inner loop iterates over a section or *strip* of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, i.e. to achieve the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's *strip length.*

In and of itself, strip mining is not profitable. However, strip mining is essential to the highly profitable loop blocking optimization, which is described later in a following section.

Loop distribution

Loop distribution transforms complicated nested loops into several simple loops. Loop distribution is a fundamental -O2 transformation that is a prerequisite to more advanced transformations that require all calculations in a nested loop to be performed inside the innermost loop.

Consider the following Fortran code.

```
DO I = 1, N
  B(I, 1) = 0
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```

Loop distribution creates three copies of the `I` loop, separating the nested `J` loop from the assignments to arrays `B` and `D`. In this way, all three assignments are moved to innermost loops, as shown in the following transformed code.

```
DO I = 1, N
  B(I, 1) = 0
ENDDO
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
ENDDO
DO I = 1, N
  D(I) = E(I) + A(I)
ENDDO
```

In and of itself, distribution is not profitable; however, it does create more opportunities for interchange.

Loop interchange

The compiler interchanges nested loops for the following reasons:

- To facilitate other transformations.
- To relocate the loop that is the most profitable to parallelize so that it is outermost (at optimization level `-O3` only).
- To optimize inner-loop memory accesses.

Consider the Fortran matrix addition algorithm that follows.

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

This loop accesses the arrays A, B and C row by row, which, in Fortran, is very inefficient. Interchanging the I and J loops, as shown below, will facilitate column by column access.

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

Loop blocking

Loop blocking is a combination of strip mining and interchange that maximizes data localization. It is provided primarily to deal with nested loops that manipulate arrays that are too large to fit into the cache. Under certain circumstances, loop blocking allows reuse of such arrays by transforming the loops that manipulate them such that they manipulate strips of the arrays that fit into the cache. Effectively, a blocked loop accesses array elements in sections that are sized to optimally fit in the cache.

Data reuse

Data reuse is important to understand when discussing blocking. There are two types of data reuse associated with loop blocking: *spatial* reuse and *temporal* reuse.

Spatial reuse is using data which was encached as a result of fetching another piece of data from memory. On an SPP Series system, data is fetched by cache lines; 32 bytes of data is encached on every fetch. On the initial fetch of array data from memory within a stride-one loop, the requested item can be located anywhere in the 32 bytes. Starting with the second memory fetch (which will not happen until any usable elements obtained on the first fetch are used), the requested data is at the beginning of the cache line, and the rest of the cache line will contain subsequent array elements. For a REAL*4 array, this means the requested element and the 7 following elements are encached on each fetch after the first. If any of these 7 elements could then be used, for instance on any subsequent iterations of the loop, the loop would be exploiting spatial reuse. For loops with strides greater than one, spatial reuse can still occur; however, the cache lines may never synch so that all the elements they contain are usable.

Temporal reuse is using the same data item on more than one iteration of the loop. An array element whose subscript does not change as a function of the iterations of a surrounding loop exhibits temporal reuse in the context of the loop.

Loops containing either temporal or spatial reuse are candidates for blocking. Blocking exploits spatial reuse by insuring that once fetched, cache lines are not overwritten until their spatial reuse is exhausted. Temporal reuse is similarly maximized.

For examples and further explanation of data reuse and loop blocking, consult the *Exemplar Programming Guide*.

Blocking directives

Loop blocking can be disabled for specific loops using the `NO_BLOCK_LOOP` compiler directive. You can advise the compiler to use a specific block factor via the `BLOCK_LOOP` directive. These directives have the following forms:

```
C$DIR NO_BLOCK_LOOP
C$DIR BLOCK_LOOP(BLOCK_FACTOR = n)
```

In the `BLOCK_LOOP` directive, *n* is the requested block factor, which must be an integer constant or constant expression. The compiler will use this value as the block factor, so for best performance, this block factor multiplied by the data type size of the data in the loop should be an integral multiple of the cache line size.

These directives affect the loop that immediately follows them. For examples of their use, refer to the *Exemplar Programming Guide*.

Loop unrolling

Loop unrolling attempts to improve efficiency by reducing or eliminating loop overhead. This is achieved by increasing a loop's step value and replicating the loop body, with each replication appropriately offset from the induction variable so that all iterations are performed given the new step.

Total unrolling involves eliminating the loop structure completely. The loop body is replicated a number of times equal to the iteration count and the iteration variable is replaced with constants. Loop overhead is completely eliminated.

Partial unrolling replicates the loop body a number of times equal to the *unroll factor* and adjusts references to the iteration variable accordingly, thus reducing the number of tests and induction variable increments.

Loop unrolling occurs at optimization levels `-O2` and higher by default and when the `-ur` compiler option is specified. Total unrolling is performed on loops with an iteration count determinable at compile time to be less than 5. Loops with

undeterminable iteration counts, or determinable counts of 5 or more, are partially unrolled.

To disable loop unrolling, specify the `-nur` compiler option. You can specify an unroll factor by using the `-urn` *n* option, where *n* is the unroll factor for all unrolled loops. Unrolling can also be requested or disabled for individual loops using the `UNROLL` and `NO_UNROLL` compiler directives, as described in *Fortran User's Guide* Appendix A.

Loop unroll and jam

The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop.

This involves partially unrolling one or more loops higher in the nest than the innermost loop, then fusing ("jamming") the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that can be temporally reused with respect to a loop other than the innermost.

Loop unroll and jam is enabled by default and through use of the `-uj` option at optimization levels `-O2` and `-O3`. It is disabled by the `-nuj` compiler option. You can specify the unrolling factor by using the `-ujn` *n* option, where *n* is the desired unrolling factor for all loops the compiler unrolls and jams.

To specify unroll and jam for individual loops use the `UNROLL_AND_JAM` compiler directive. Because unroll and jam is performed only on nested loops, you must ensure that the `UNROLL_AND_JAM` directive is specified for a loop that, after any compiler-initiated interchanges, ends up being non-innermost. You can determine which loop in a nest will be innermost by compiling the nest without any directives and examining the optimization report.

The `NO_UNROLL_AND_JAM` directive disables unroll and jam for an individual loop. Both directives are covered in *Fortran User's Guide* Appendix A and in the *Exemplar Programming Guide.*

`IF-DO` optimizations

These optimizations modify loops containing tests to improve performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of branches that must be executed, thereby improving performance.

In Fortran, these optimizations are performed on DO loops containing IF statements. These optimizations fall into one of three categories: redundant test elimination, loop peeling, or test promotion. Each of these is described below.

Redundant-test elimination

Redundant-test elimination is the simplest of these optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code.

This optimization is especially relevant when you are optimizing FORTRAN 66 programs that contain DO loops surrounded by IF tests, as shown in the following example.

```
DO I = 1, N
  IF (I .GT. 0) THEN
    DO J = 1, I
      A(I,J) = 0
    ENDDO
  ENDIF
ENDDO
```

With the test removed, the loop looks like this:

```
DO I = 1, N
  DO J = 1, I
    A(I,J) = 0
  ENDDO
ENDDO
```

Here the explicit test IF (I .GT. 0) is redundant, since the test is implicit in the DO loop. It is therefore removed during redundant-test elimination.

Redundant-test elimination is always performed at optimization levels -O2 and above.

Loop boundary-value peeling

Loop boundary-value peeling involves removing the first iteration, last iteration, or first and last iterations of a loop to remove conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates identically for the first iteration, last iteration, or first and last iterations.

Given the Fortran code shown below, the compiler automatically peels off the first and last tests and rewrites the loop to cover the remaining indexes.

```
DO I = 1, 100
  IF (I .EQ. 1) THEN
    A(I) = B(I)
  ELSE IF (I .EQ. 100) THEN
    A(I) = C(I)
  ELSE
    A(I) = -A(I)
  ENDIF
ENDDO
```

After peeling, the code looks like this:

```
A(1) = B(1)
DO I = 2, 99
  A(I) = -A(I)
ENDDO
A(100) = C(100)
```

In some cases, boundary-value peeling requires replicating large amounts of code and can greatly increase the size of the executable file. By default, the compiler peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or, if you wish to do so on a loop-by-loop basis, the `PEEL` compiler directive.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option or the `PEEL_ALL` directive. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL` compiler directive.

Loop boundary-value peeling is not performed on loops that have no tests on boundary values. In other words, the compiler does not try to peel unpeelable loops.

Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.

Consider the following Fortran loop.

```
DO I = 1, N
  IF (FOO .EQ. BAR) THEN
    A(I) = B(I)
  ELSE
    A(I) = 0
  ENDIF
ENDDO
```

Test promotion produces the following code.

```
IF (FOO .EQ. BAR) THEN
  DO I = 1, N
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1, N
    A(I) = 0
  ENDDO
ENDIF
```

For loops containing large numbers of tests, loop replication can greatly increase the size of the code.

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler promotes tests and replicates code up to a predetermined conservative limit.

The `-ptst` compiler option increases this limit and can cause a noticeable increase in compile time.

The `-ptstall` option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

The `-noptst` option disables test promotion.

The `PROMOTE_TEST`, `PROMOTE_TEST_ALL` and `NO_PROMOTE_TEST` compiler directives provide similar functionality on a loop-by-loop basis.

At optimization levels `-O2` and above, the compiler automatically performs these optimizations on `DO` and hand-rolled loops. Simple and nested loops, and loops with exits are handled. For more information refer to the *Exemplar Programming Guide*.

Scalar replacement

Scalar replacement involves storing loop-invariant array elements in registers. This substantially increases the loop's performance by eliminating the need to access the array element in main memory or one of the caches.

CONVEX Fortran will not attempt scalar replacement if the loop-invariant array element is aliased or if the array element is contained in conditional code.

Scalar replacement is enabled by default and via the `-sr` compiler option at optimization levels `-O2` and `-O3`. To disable scalar replacement specify the `-nsr` option.

Preventing data localization

The `SCALAR` directive allows you to prevent all loop-reordering transformations on the immediately following loop. This directive has the following form:

```
C$DIR SCALAR
```

-O3 Level optimizations

The primary goal of `-O3` optimizations is *parallelization*. Parallelization divides a program into threads. A *thread* is a sequence of instructions that must execute on a single CPU.

The SPP Series Fortran compiler finds parallelism at the loop level and generates parallel code that will automatically run on as many processors as are available at runtime. Normally, these are all the processors of the subcomplex on which your program is running. You can specify a smaller number of processors via the `mpa` utility. Refer to the mpa(1) man page for more information. You can also specify a smaller number of processors using the `LOOP_PARALLEL(MAX_THREADS=`*m*`)` or `PREFER_PARALLEL(MAX_THREADS=`*m*`)` compiler directives, as described in *Fortran User's Guide* Appendix A.

Basic operation

Parallelism can exist at both the loop level and the task level. The CONVEX SPP Series Fortran compiler automatically exploits loop-level parallelism. You can easily identify task-level parallelism using the `BEGIN_TASKS`, `NEXT_TASK` and `END_TASKS` directives, as discussed in *Fortran User's Guide* Appendix A. These directives allow the compiler to run identified sections of code in parallel.

Loop-level parallelism involves dividing a loop up into several smaller iteration spaces and parceling these out to be run simultaneously on all available processors.

Consider the following Fortran code.

```
DO I = 1, 1024
  A(I) = B(I) + C(I)
ENDDO
```

This code can be parallelized to run on 8 processors by running 128 iterations per processor (1024 iterations divided by 8 processors = 128 iterations each). One processor would run the loop for `I` = 1 to 128; the next would run `I` = 129 to 256, and so on. The loop could similarly be parallelized to run on any number of processors, with each one taking its appropriate share of iterations. The compiler generates code that will run on as many processors as are available. If the number of available processors does not evenly divide the number of iterations, some processors will perform fewer iterations than others.

For a more detailed explanation of SPP Series `-O3` optimizations, consult the *Exemplar Programming Guide.*

Fortran 90 constructs

Fortran 90 array expressions are translated into loops by the compiler, and parallelism in these loops will be automatically exploited. For example, the following Fortran 90 statement:

```
X(1:M:2, 1:N) = Y(2:M+1:2,2:N+1)
```

is translated by the compiler into a loop similar to the following:

```
DO I = 1, M, 2
  DO J = 1, N
    X(I,J) = Y(I+1,J+1)
  ENDDO
ENDDO
```

which then can be automatically parallelized.

Masked array assignments are similarly parallelized. Consider the following WHERE statement.

```
REAL DATA(1000), LIMIT
LOGICAL NORMAL(1000)
.
.
.
WHERE(DATA .LE. LIMIT) NORMAL = .TRUE.
```

The compiler translates the WHERE statement into a loop similar to the following:

```
DO I = 1, 1000
  IF(DATA(I) .LE. LIMIT) NORMAL(I) = .TRUE.
ENDDO
```

which then can be automatically parallelized.

Parallel optimizations

At optimization level -O3, parallelism is, in effect, *the* optimization. All -O2 level optimizations are also performed, but parallelization is the only optimization added at -O3.

Simple loops can be parallelized without the need for extensive -O2 transformations, as shown in the "Basic operation" section. However, most loop transformations, if they are applicable to the loop in question, can aid parallelization in some way. For instance, loop interchange orders loops such that the innermost loop best exploits the processor data cache, and the outermost loop is the most efficient loop to parallelize. Loop blocking similarly aids parallelization by maximizing cache data reuse on each of the processors that the loop runs on, and by insuring that each processor is working on non-overlapping array data.

Preventing parallelization

Parallelization can be disabled on a loop basis by using the NO_PARALLEL directive, which has the following form:

```
C$DIR NO_PARALLEL
```

You can use this directive to prevent parallelization of the loop that immediately follows it. Only parallelization is inhibited; all other loop optimizations will still be applied. The following Fortran example illustrates the use of this directive.

```
      DO I = 1, 1000
C$DIR   NO_PARALLEL
        DO J = 1, 1000
          A(I,J) = B(I,J)
        ENDDO
      ENDDO
```

In this example, parallelization of the J loop is prevented. The I loop still can be parallelized.

Other parallelization directives

Several directives are available to allow you to manually control certain aspects of loop parallelization, and to parallelize tasks outside of loops. These directives are:

- PREFER_PARALLEL—requests parallelization of the immediately following loop; accepts attributes for node- and thread-parallelism, strip-length adjustment, maximum number of threads, and ordered execution. The compiler handles data privatization and will not parallelize the loop if it is not safe to do so.
- LOOP_PARALLEL—forces parallelization of the immediately following loop. Accepts the same attributes as PREFER_PARALLEL, but requires you to manually privatize loop data and synchronize data dependencies.
- BEGIN_TASKS, NEXT_TASK, and END_TASKS—allow you to parallelize regions of code outside of loops. Accepts attributes for node- and thread-parallelism, ordered execution, and maximum number of threads.
- CRITICAL_SECTION, END_CRITICAL_SECTION—allow you to isolate non-ordered manipulations of a shared variable within the loop. Only one parallel thread can execute the code contained in the critical section at a time, eliminating possible contention.
- ORDERED_SECTION, END_ORDERED_SECTION—allow you to isolate dependencies within a loop so that code contained within the ordered section executes in iteration order.

These directives are discussed in detail in *Fortran User's Guide* Appendix A and in the *Exemplar Programming Guide*.

Intrinsics and commonly used library routines

A

This appendix lists CONVEX Fortran's generic and specific intrinsics, as well as its commonly used library routines.

Generic and specific intrinsics

Table 35 lists generic and specific intrinsics. Numbers in the first and second column of Table 35refer to notes following the table.

All intrinsic routines *except* those with an asterisk (*) footnote mark can be used in array-valued assignment statements. Intrinsic routines marked with a diamond (✦) footnote mark are available only on CONVEX SPP Series machines. CONVEX extensions to the language are set in italic type.

Table 35 Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
SQRT	SQRT	Square root $a^{1/2}$	1	REAL*4	REAL*4
1	DSQRT			REAL*8	REAL*8
	QSQRT			*REAL*16*	*REAL*16*
	CSQRT			COMPLEX*8	COMPLEX*8
	CDSQRT			*COMPLEX*16*	*COMPLEX*16*
LOG	ALOG	Natural log $\log_e a$	1	REAL*4	REAL*4
2	DLOG			REAL*8	REAL*8
	QLOG			*REAL*16*	*REAL*16*
	CLOG			COMPLEX*8	COMPLEX*8
	CDLOG			*COMPLEX*16*	*COMPLEX*16*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
LOG10	ALOG10	Common log $\log_{10} a$	1	REAL*4	REAL*4
2	DLOG10			REAL*8	REAL*8
	QLOG10			*REAL*16*	*REAL*16*
EXP	EXP	Exponential e^a	1	REAL*4	REAL*4
	DEXP			REAL*8	REAL*8
	QEXP			*REAL*16*	*REAL*16*
	CEXP			COMPLEX*8	COMPLEX*8
	CDEXP			*COMPLEX*16*	*COMPLEX*16*
SIN	SIN	Sine $\sin a$	1	REAL*4	REAL*4
3	DSIN			REAL*8	REAL*8
	QSIN			*REAL*16*	*REAL*16*
	CSIN			COMPLEX*8	COMPLEX*8
	CDSIN			*COMPLEX*16*	*COMPLEX*16*
SIND	*SIND*	*Sine (degree)* $\sin a^\circ$	1	REAL*4	REAL*4
3	*DSIND*			REAL*8	REAL*8
	QSIND			*REAL*16*	*REAL*16*
COS	COS	Cosine $\cos a$	1	REAL*4	REAL*4
3	DCOS			REAL*8	REAL*8
	QCOS			*REAL*16*	*REAL*16*
	CCOS			COMPLEX*8	COMPLEX*8
	CDCOS			*COMPLEX*16*	*COMPLEX*16*
COSD	*COSD*	*Cosine (degree)* $\cos a^\circ$	1	REAL*4	REAL*4
3	*DCOSD*			REAL*8	REAL*8
	QCOSD			*REAL*16*	*REAL*16*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
TAN	TAN	Tangent tan *a*	1	REAL*4	REAL*4
3	DTAN			REAL*8	REAL*8
	QTAN			*REAL*16*	*REAL*16*
TAND	*TAND*	*Tangent (degree)* *tan a°*	1	REAL*4	REAL*4
3	*DTAND*			REAL*8	REAL*8
	QTAND			*REAL*16*	*REAL*16*
ASIN	ASIN	Arcsine arcsin *a*	1	REAL*4	REAL*4
4,5	DASIN			REAL*8	REAL*8
	QASIN			*REAL*16*	*REAL*16*
ASIND	*ASIND*	*Arcsine (degree)* *arcsin a*	1	REAL*4	REAL*4
2,4,5	*DASIND*			REAL*8	REAL*8
	QASIND			*REAL*16*	*REAL*16*
ACOS	ACOS	Arccosine arccos *a*	1	REAL*4	REAL*4
4,5	DACOS			REAL*8	REAL*8
	QACOS			*REAL*16*	*REAL*16*
ACOSD	*ACOSD*	*Arccosine (degree)* *arccos a*	1	REAL*4	REAL*4
2,4,5	*DACOSD*			REAL*8	REAL*8
	QACOSD			*REAL*16*	*REAL*16*
ATAN	ATAN	Arctangent arctan *a*	1	REAL*4	REAL*4
5	DATAN			REAL*8	REAL*8
	QATAN			*REAL*16*	*REAL*16*
ATAND	*ATAND*	*Arctangent (degree)* *arctan a*	1	REAL*4	REAL*4
2,5	*DATAND*			REAL*8	REAL*8
	QATAND			*REAL*16*	*REAL*16*

Table 35 (continued)Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
ATAN2	ATAN2	Arctangent (two arguments) arctan a_1/a_2	2	REAL*4	REAL*4
5,6	DATAN2			REAL*8	REAL*8
	QATAN2			*REAL*16*	*REAL*16*
ATAN2D	*ATAN2D*	*Arctangent (two degree arguments) Arctan a_1 °/a_2 °*	2	REAL*4	REAL*4
2,5,7	*DATAN2D*			REAL*8	REAL*8
	QATAN2D			*REAL*16*	*REAL*16*
SINH	SINH	Hyperbolic Sine sinh a	1	REAL*4	REAL*4
	DSINH			REAL*8	REAL*8
	QSINH			*REAL*16*	*REAL*16*
COSH	COSH	Hyperbolic Cosine cosh a	1	REAL*4	REAL*4
	DCOSH			REAL*8	REAL*8
	QCOSH			*REAL*16*	*REAL*16*
TANH	TANH	Hyperbolic Tangent tanh a	1	REAL*4	REAL*4
	DTANH			REAL*8	REAL*8
	QTANH			*REAL*16*	*REAL*16*
ABS	*IIABS*	Absolute value $\|a\|$	1	*INTEGER*2*	*INTEGER*2*
8	*JIABS*			INTEGER*4	INTEGER*4
	KIABS			*INTEGER*8*	*INTEGER*8*
	ABS			REAL*4	REAL*4
	DABS			REAL*8	REAL*8
	QABS			*REAL*16*	*REAL*16*
	CABS			COMPLEX*8	REAL*4
	CDABS			*COMPLEX*16*	*REAL*8*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
IABS	*IIABS*	*Absolute value* *|a|*	1	*INTEGER*2*	*INTEGER*2*
8	*JIABS*			INTEGER*4	INTEGER*4
	KIABS			*INTEGER*8*	*INTEGER*8*
INT	*IINT*	*Truncation* *[a]*	1	REAL*4	*INTEGER*2*
9,14	*JINT*			REAL*4	INTEGER*4
15	*KINT*			REAL*4	*INTEGER*8*
	IIDINT			REAL*8	*INTEGER*2*
	JIDINT			REAL*8	INTEGER*4
	KIDINT			REAL*8	*INTEGER*8*
	IIQINT			*REAL*16*	*INTEGER*2*
	JIQINT			*REAL*16*	INTEGER*4
	KIQINT			*REAL*16*	*INTEGER*8*
INT				COMPLEX*8	*INTEGER*2*
19				COMPLEX*8	INTEGER*4
				COMPLEX*8	*INTEGER*8*
				*COMPLEX*16*	*INTEGER*2*
				*COMPLEX*16*	INTEGER*4
				*COMPLEX*16*	*INTEGER*8*
10,14	*INT1*	*Conversion to INTEGER*	1	*Any numeric*	*INTEGER*1*
	INT2			*Any numeric*	*INTEGER*2*
	INT4			*Any numeric*	INTEGER*4
	INT8			*Any numeric*	*INTEGER*8*

Table 35 (continued)Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
IDINT	*IIDINT*	*Truncation [a]*	1	REAL*8	*INTEGER*2*
9,14	*JIDINT*			REAL*8	INTEGER*4
15	*KIDINT*			REAL*8	*INTEGER*8*
IQINT	*IIQINT*	*Truncation [a]*	1	*REAL*16*	*INTEGER*2*
9,14	*JIQINT*			*REAL*16*	INTEGER*4
15	*KIQINT*			*REAL*16*	*INTEGER*8*
AINT	AINT	*Truncation [a]*	1	REAL*4	REAL*4
	DINT			REAL*8	REAL*8
	QINT			*REAL*16*	*REAL*16*
NINT	*ININT*	*Nearest integer [a+.5*sign(a)]*	1	REAL*4	*INTEGER*2*
9,14	*JNINT*			REAL*4	INTEGER*4
15	*KNINT*			REAL*4	*INTEGER*8*
	IIDNNT			REAL*8	*INTEGER*2*
	JIDNNT			REAL*8	INTEGER*4
	KIDNNT			REAL*8	*INTEGER*8*
	IIQNNT			*REAL*16*	*INTEGER*2*
	JIQNNT			*REAL*16*	INTEGER*4
	KIQNNT			*REAL*16*	*INTEGER*8*
IDNINT	*IIDNNT*	*Nearest integer [a+.5*sign(a)]*	1	REAL*8	*INTEGER*2*
9,14	*JIDNNT*			REAL*8	INTEGER*4
15	*KIDNNT*			REAL*8	*INTEGER*8*
IQNINT	*IIQNNT*	*Nearest integer [a+.5*sign(a)]*	1	*REAL*16*	*INTEGER*2*
9,14	*JIQNNT*			*REAL*16*	INTEGER*4
15	*KIQNNT*			*REAL*16*	*INTEGER*8*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
ANINT	ANINT	Nearest integer [*a*+.5*sign(*a*)]	1	REAL*4	REAL*4
9,15	DNINT			REAL*8	REAL*8
	QNINT			*REAL*16*	*REAL*16*
*ZEXT**	*IZEXT*	*Zero-extend a*	1	*LOGICAL*1*	*INTEGER*2*
14,15	*19*			*LOGICAL*2*	*INTEGER*2*
	19			*INTEGER*2*	*INTEGER*2*
	JZEXT			*LOGICAL*1*	INTEGER*4
	19			*LOGICAL*2*	INTEGER*4
	19			LOGICAL*4	INTEGER*4
	19			INTEGER*2	INTEGER*4
	19			INTEGER*4	INTEGER*4
	KZEXT			*LOGICAL*1*	*INTEGER*8*
	19			*LOGICAL*2*	*INTEGER*8*
	19			LOGICAL*4	*INTEGER*8*
	19			LOGICAL*8	*INTEGER*8*
	19			*INTEGER*2*	*INTEGER*8*
	19			INTEGER*4	*INTEGER*8*
	19			*INTEGER*8*	*INTEGER*8*

Table 35 (continued)Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
REAL	*FLOATI*	Convert *a* to *REAL*.	1	*INTEGER*2*	REAL
10	*FLOATJ*	*By default, return REAL*4.*		INTEGER*4	REAL
	FLOATK	*When the* -cfc, -p8, *or* -pd8 *compiler option is specified, return REAL*8.*		*INTEGER*8*	REAL
	19			REAL*4	REAL
	SNGL			REAL*8	REAL
	19			COMPLEX*8	REAL
	19			*COMPLEX*16*	REAL
	SNGLQ			*REAL*16*	REAL
DBLE	*19*	Convert *a* to REAL*8	1	*INTEGER*2*	REAL*8
10	*19*			INTEGER*4	REAL*8
	19			*INTEGER*8*	REAL*8
	19			REAL*4	REAL*8
	19			REAL*8	REAL*8
	19			COMPLEX*8	REAL*8
	19			*COMPLEX*16*	REAL*8
	DBLEQ			*REAL*16*	REAL*8

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
QEXT		*Convert a to REAL*16*	1	*INTEGER*2*	*REAL*16*
19				INTEGER*4	*REAL*16*
				*INTEGER*8*	*REAL*16*
				REAL*4	*REAL*16*
	QEXTD			REAL*8	*REAL*16*
	19			*REAL*16*	*REAL*16*
	19			COMPLEX*8	*REAL*16*
	19			*COMPLEX*16*	*REAL*16*
IFIX	*IIFIX*	*Fix a (REAL*4-to-INTEGER conversion)*	1	REAL*4	*INTEGER*2*
10,14	*JIFIX*			REAL*4	INTEGER*4
15	*KIFIX*			REAL*4	*INTEGER*8*
FLOAT	*FLOATI*	*Float a (INTEGER-to-REAL*4 conversion)*	1	*INTEGER*2*	REAL*4
10	*FLOATJ*			INTEGER*4	REAL*4
	FLOATK			*INTEGER*8*	REAL*4
DFLOAT	*DFLOTI*	*Float a (INTEGER-to-REAL*8 conversion)*	1	*INTEGER*2*	REAL*8
10	*DFLOTJ*			INTEGER*4	REAL*8
	DFLOTK			*INTEGER*8*	REAL*8
QFLOAT	*QFLOTI*	*Float a (INTEGER-to-REAL*16 conversion)*	1	*INTEGER*2*	*REAL*16*
10	*QFLOTJ*			INTEGER*4	*REAL*16*
	QFLOTK			*INTEGER*8*	*REAL*16*

Table 35 (continued)Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
CMPLX		Convert a_1 to COMPLEX*8 or convert a_1 and a_2 to COMPLEX*8	1,2	*INTEGER*2*	COMPLEX*8
19			1,2	INTEGER*4	COMPLEX*8
			1,2	*INTEGER*8*	COMPLEX*8
			1,2	REAL*4	COMPLEX*8
			1,2	REAL*8	COMPLEX*8
			1	COMPLEX*8	COMPLEX*8
			1	*COMPLEX*16*	COMPLEX*8
DCMPLX		*Convert a_1 to COMPLEX*16 or convert a_1 and a_2 to COMPLEX*16*	1,2	*INTEGER*2*	*COMPLEX*16*
11,15			1,2	INTEGER*4	*COMPLEX*16*
19			1,2	*INTEGER*8*	*COMPLEX*16*
			1,2	REAL*4	*COMPLEX*16*
			1,2	REAL*8	*COMPLEX*16*
			1	COMPLEX*8	*COMPLEX*16*
			1	*COMPLEX*16*	*COMPLEX*16*
	REAL	Real part of complex	1	COMPLEX*8	REAL*4
	DREAL			*COMPLEX*16*	REAL*8
	AIMAG	Imaginary part of complex	1	COMPLEX*8	REAL*4
	DIMAG			*COMPLEX*16*	REAL*8
CONJG	CONJG	Complex conjugate (if $a=(x,y)$ then CONJG$(a)=(x,-y)$)	1	COMPLEX*8	COMPLEX*8
	DCONJG			*COMPLEX*16*	*COMPLEX*16*
	DPROD	REAL*8 product of REAL*4 a_1*a_2	2	REAL*4	REAL*8

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
MAX*	*IMAX0*	Maximum max(a_1, a_2,...a_n)	*n*	*INTEGER*2*	*INTEGER*2*
12,15	*JMAX0*			INTEGER*4	INTEGER*4
	KMAX0			INTEGER*8	INTEGER*8
	AMAX1			REAL*4	REAL*4
	DMAX1			REAL*8	REAL*8
	IIDMAX1			REAL*8	*INTEGER*2*
	JIDMAX1			REAL*8	INTEGER*4
	KIDMAX1			REAL*8	INTEGER*8
	QMAX1			*REAL*16*	*REAL*16*
*MAX0**	*IMAX0*	*Maximum max(a_1,a_2,...a_n)*	*n*	*INTEGER*2*	*INTEGER*2*
12,15	*JMAX0*			INTEGER*4	INTEGER*4
	KMAX0			*INTEGER*8*	*INTEGER*8*
*MAX1**	*IMAX1*	*Maximum max(a_1,a_2,...a_n)*	*n*	REAL*4	*INTEGER*2*
12,14	*JMAX1*			REAL*4	INTEGER*4
15	*KMAX1*			REAL*4	*INTEGER*8*
*AMAX0**	*AIMAX0*	*Maximum max(a_1,a_2,...a_n)*	*n*	*INTEGER*2*	REAL*4
12,15	*AJMAX0*			INTEGER*4	REAL*4
	AKMAX0			*INTEGER*8*	REAL*4

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
MIN*	*IMIN0*	Minimum min(a_1,a_2,...a_n)	*n*	*INTEGER*2*	*INTEGER*2*
13,15	*JMIN0*			INTEGER*4	INTEGER*4
	KMIN0			*INTEGER*8*	*INTEGER*8*
	AMIN1			REAL*4	REAL*4
	DMIN1			REAL*8	REAL*8
	IIDMIN1			REAL*8	*INTEGER*2*
	JIDMIN1			REAL*8	INTEGER*4
	KIDMIN1			REAL*8	*INTEGER*8*
	QMIN1			*REAL*16*	*REAL*16*
*MIN0**	*IMIN0*	*Minimum min(a_1,a_2,...a_n)*	*n*	*INTEGER*2*	*INTEGER*2*
13,15	JMIN0			INTEGER*4	INTEGER*4
	KMIN0			*INTEGER*8*	*INTEGER*8*
*MIN1**	*IMIN1*	*Minimum min(a_1,a_2,...a_n)*	*n*	REAL*4	*INTEGER*2*
13,14	*JMIN1*			REAL*4	INTEGER*4
15	*KMINI*			REAL*4	*INTEGER*8*
*AMIN0**	*AIMIN0*	*Minimum min(a_1,a_2,...a_n)*	*n*	*INTEGER*2*	REAL*4
	AJMIN0			INTEGER*4	REAL*4
DIM	*IIDIM*	Positive difference a1- (min(a1,a2))	2	*INTEGER*2*	*INTEGER*2*
	JIDIM			INTEGER*4	INTEGER*4
	KIDIM			*INTEGER*8*	*INTEGER*8*
	DIM			REAL*4	REAL*4
	DDIM			REAL*8	REAL*8
	QDIM			*REAL*16*	*REAL*16*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
IDIM	*IIDIM*	Positive difference a_1- (min(a_1 , a_2))	2	*INTEGER*2*	*INTEGER*2*
	JIDIM			INTEGER*4	INTEGER*4
	KIDIM			*INTEGER*8*	*INTEGER*8*
MOD	*IMOD*	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*2*
	JMOD			INTEGER*4	INTEGER*4
	KMOD			*INTEGER*8*	*INTEGER*8*
	AMOD			REAL*4	REAL*4
	DMOD			REAL*8	REAL*8
	QMOD			*REAL*16*	*REAL*16*
SIGN	*IISIGN*	Transfer of sign $\lvert a_1 \rvert$ if $a_2 \geq 0$ $-\lvert a_1 \rvert$ if $a_2 < 0$	2	*INTEGER*2*	*INTEGER*2*
	JISIGN			INTEGER*4	INTEGER*4
	KISIGN			*INTEGER*8*	*INTEGER*8*
	SIGN			REAL*4	REAL*4
	DSIGN			REAL*8	REAL*8
	QSIGN			*REAL*16*	*REAL*16*
ISIGN	*IISIGN*	Transfer of sign $\lvert a_1 \rvert$ sign a_2	2	*INTEGER*2*	*INTEGER*2*
	JISIGN			INTEGER*4	INTEGER*4
	KISIGN			*INTEGER*8*	*INTEGER*8*
IAND	*IIAND*	*Bitwise AND (performs a logical AND on corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIAND			INTEGER*4	INTEGER*4
	KIAND			*INTEGER*8*	*INTEGER*8*
IOR	*IIOR*	*Bitwise OR (performs an inclusive OR on corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIOR			INTEGER*4	INTEGER*4
	KIOR			*INTEGER*8*	*INTEGER*8*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	**Specific**				
IEOR	*IIEOR*	*Bitwise XOR (exclusively ORs corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIEOR			INTEGER*4	INTEGER*4
	KIEOR			*INTEGER*8*	*INTEGER*8*
NOT	*INOT*	*Bitwise complement (complements each bit)*	1	*INTEGER*2*	*INTEGER*2*
	JNOT			INTEGER*4	INTEGER*4
	KNOT			*INTEGER*8*	*INTEGER*8*
ISHFT†	IISHFT	*Bitwise shift (a1 logically shifted a2 bits—positive a2 argument shifts left; negative, right)*	2	*INTEGER*2*	*INTEGER*2*
16	*JISHFT*			INTEGER*4	INTEGER*4
	KISHFT			*INTEGER*8*	*INTEGER*8*
IBITS†	*IIBITS*	*Bit extraction (extracts bits a2 through a2+a3-1 from a1)*	3	*INTEGER*2*	*INTEGER*2*
17	*JIBITS*			INTEGER*4	INTEGER*4
	KIBITS			*INTEGER*8*	*INTEGER*8*
IBSET†	*IIBSET*	*Bit set (returns the value of a1 with bit a2 of a1 set to 1)*	2	*INTEGER*2*	*INTEGER*2*
	JIBSET			INTEGER*4	INTEGER*4
	KIBSET			*INTEGER*8*	*INTEGER*8*
BTEST†	*BITEST*	*Bit test (returns TRUE if bit a2 of argument a1 equals 1)*	2	*INTEGER*2*	*LOGICAL*2*
	BJTEST			INTEGER*4	LOGICAL*4
	BKTEST			*INTEGER*8*	*LOGICAL*8*
IBCLR†	*IIBCLR*	*Bit clear (returns the value of a1 with bit a2 of a1 set to 0)*	2	*INTEGER*2*	*INTEGER*2*
	JIBCLR			INTEGER*4	INTEGER*4
	KIBCLR			*INTEGER*8*	*INTEGER*8*
ISHFTC†	*IISHFTC*	*Bitwise circular shift (circularly shifts rightmost a3 bits of argument a1 by a2)*	3	*INTEGER*2*	*INTEGER*2*
	JISHFTC			INTEGER*4	INTEGER*4
	KISHFTC			*INTEGER*8*	*INTEGER*8*

Table 35 (continued) Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
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	INTEGER*4
15 19	CHAR	Character (returns a character that has the ASCII value specified by the argument)	1	*LOGICAL*1* *INTEGER*2* INTEGER*4 *INTEGER*8*	CHARACTER CHARACTER CHARACTER CHARACTER
15	ICHAR	ASCII value (returns the ASCII value of the argument, which must be a character expression of length 1)	1	CHARACTER	INTEGER*4
	LLT	Lexical comparison (TRUE if string *c1* precedes string *c2* in the ASCII collating sequence)	2	CHARACTER	LOGICAL
	LLE	Lexical comparison (TRUE if string *c1* precedes or equals string *c2* in the ASCII collating sequence)	2	CHARACTER	LOGICAL
	LGT	Lexical comparison (TRUE if string *c1* follows string *c2* in the ASCII collating sequence)	2	CHARACTER	LOGICAL
	LGE	Lexical comparison (TRUE if string *c1* follows or equals string *c2* in the ASCII collating sequence)	2	CHARACTER	LOGICAL

Table 35 (continued) Generic and specific intrinsics

Intrinsics Generic	Function	No. args	Argument type	Result type
DOT_PRODUCT* 18	Dot product of two rank-one arrays	2	Varies‡	Varies‡
MATMUL* 18	Matrix multiply	2	Varies‡	Varies‡
ALL* 18	True if all array elements along optional dimension are true	1,2	Varies‡	Varies‡
ANY* 18	True if any array element along optional dimension is true	1,2	Varies‡	Varies‡
COUNT* 18	Counts number of true elements along optional dimension in array	1,2	Varies‡	Varies‡
MAXVAL* 18	Maximum value along optional dimension using optional mask	1-3	Varies‡	Varies‡
MINVAL* 18	Minimum value along optional dimension using optional mask	1-3	Varies‡	Varies‡
PRODUCT* 18	Product of array elements along optional dimension using optional mask	1-3	Varies‡	Varies‡
SUM 18	Sum all the elements of an an array along optional dimension, with an optional mask	1-3	Varies‡	Varies‡
MERGE 18	Merge under mask	3	Varies‡	Varies‡
PACK* 18	Pack an array into an array of rank one under mask	3	Varies‡	Varies‡
SPREAD* 18	Replicates array by adding a dimension	3	Varies‡	Varies‡

Table 35 (continued) Generic and specific intrinsics

Intrinsics Generic	Function	No. args	Argument type	Result type
UNPACK✳ *18*	*Unpack an array of rank one into an array under a mask*	3	*Varies*‡	*Varies*‡
CSHIFT✳ *18*	*Circular shift array elements*	2-3	*Varies*‡	*Varies*‡
EOSHIFT✳ *18*	*End-off shift array elements*	2-3	*Varies*‡	*Varies*‡
TRANSPOSE✳ *18*	*Transpose of an array of rank two*	1	*Varies*‡	*Varies*‡
MAXLOC✳ *18*	*Location of a maximum value in an array under an optional mask*	2,3	*Varies*‡	*Varies*‡
MINLOC✳ *18*	*Location of a minimum value in an array under an optional mask*	2,3	*Varies*‡	*Varies*‡
NUM_PROCS✦	*Return the number of processors the process is using at runtime.*	0		*INTEGER*
NUM_PROCS_8✦		0		*INTEGER*8*
NUM_THREADS✦	*Return the number of threads the process is using at runtime.*	0		*INTEGER*
NUM_THREADS_8✦		0		*INTEGER*8*
NUM_NODES✦	*Return the number of nodes on which the process is running.*	0		*INTEGER*
NUM_NODES_8✦		0		*INTEGER*8*
MY_THREAD✦	*Return the spawn thread ID.*	0		*INTEGER*
MY_THREAD_8✦		0		*INTEGER*8*
MY_NODE✦	*Return the node ID of the calling thread.*	0		*INTEGER*
MY_NODE_8✦		0		*INTEGER*8*
NUM_NODE_THREADS✦	*Return the number of threads on the node the calling threads is on.*	0		*INTEGER*
NUM_NODE_THREADS_8✦		0		*INTEGER*8*

Table 35 **(continued)**Generic and specific intrinsics

Intrinsics Generic	Function	No. args	Argument type	Result type
LEVEL_OF_PARALLELISM✦	*Return the level of parallelism of the current process.*	0		*INTEGER*
LEVEL_OF_PARALLELISM_8✦		0		*INTEGER*8*
MEMORY_TYPE_OF_STACK✦	*Return the memory class the current thread stack is allocated from.*	0		*INTEGER*
MEMORY_TYPE_OF_STACK_8✦		0		*INTEGER*8*

* All intrinsics *except these* return array results when called with array arguments.
† Arguments after the first are converted to the type of the first argument.
‡ The argument list to the corresponding library routines is variable for these Fortran 90 intrinsics. The types of the arguments to these functions are not predetermined.
✦ These intrinsic routines are available only on CONVEX SPP Series machines.

Notes

1. The SQRT, DSQRT, or QSQRT 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 and the imaginary part greater than or equal to zero.

2. The ALOG, DLOG, QLOG, ALOG10, DLOG10, QLOG10, ATAND, ATAN2D, ASIND, DASIND, QASIND, ACOSD, DACOSD, or QACOSD argument, must be greater than zero. The CLOG or CDLOG argument cannot be (0.,0.).

3. The SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, or QTAN argument must be in radians, and is treated as modulo 2π. The SIND, COSD, or TAND argument must be in degrees; the argument is treated as modulo 360. The value of the sine and cosine functions for very large arguments is not meaningful because of the accuracy of the argument reduction. For single precision, the maximum value is $\pi*2^{23}$; for double precision, the maximum value is $\pi*2^{52}$; for quadruple precision, the maximum value is $\pi*2^{112}$. If the argument exceeds the maximum, it is replaced with 0 and evaluation continues.

4. The absolute value of the ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ASIND, DASIND, QASIND, ACOSD, DACOSD, or QACOSD argument must be less than or equal to 1.

5. The result of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ATAN, DATAN, QATAN, ATAN2, DATAN2, or QATAN2 is in

radians, and that of ASIND, DASIND, QASIND, ACOSD, DACOSD, QACOSD, ATAND, DATAND, QATAND, ATAN2D, DATAN2D, or QATAN2D is in degrees.

6. If the value of the first ATAN2, DATAN2, or QATAN2 argument is positive, the result is positive; if it is zero, the result is zero if the second argument is positive, and π if the second argument 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 $\pi/2$. Both arguments cannot have the value zero. The range of the result for ATAN2, DATAN2, or QATAN2 is $-\pi \leq \text{result} < \pi$.

7. If the value of the first ATAN2D, DATAN2D, or QATAN2D 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 the second argument 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. Both arguments cannot be zero. The range of the result for ATAN2, DTAN2D, or QATAN2D 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, IQINT, NINT, IDNINT, IQNINT, IFIX, MAX1, MINI, and ZEXT functions return INTEGER*4 values if the /I4 (COVUE only) or -i4 option is in effect, *INTEGER*2* values if the */NOI4* (COVUE only) or *-i2* option is in effect, or *INTEGER*8* values if the *-i8*, *-p8*, or *-pd8* option is in effect.

16. These functions shift binary patterns—positive, left (a_1 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. In CONVEX Fortran, bits within a word are numbered from right to left. For example, in REAL*4 data, the rightmost (least significant) bit is 0; the leftmost (most significant) bit is 31.

    Bit extraction proceeds from right to left; thus, a_2 is the rightmost bit extracted and a_2+a_3-1 is the leftmost bit. The extracted field is shifted right before being placed into the receiving field. If the extracted field is shorter than the receiving field, zeros are filled in from the left.

    The following example illustrates bit extraction:

```
      PROGRAM TEST
      INTEGER*4 I1, J1, K1, L1
      I1 = 'FFFFFFFF'X

      J1 = IBITS (I1,1,4)
      K1 = IBITS (I1,29,4)
      L1 = IBITS (I1,16,6)

      WRITE (6,100) 'I1 = ',I1
      WRITE (6,100) 'J1 = ',J1
      WRITE (6,100) 'K1 = ',K1
      WRITE (6,100) 'L1 = ',L1

  100 FORMAT (A4,Z8)
      END
```

    When the program is executed, the output displays the original bit pattern and the extracted bit fields, as follows:

```
 I1 =FFFFFFFF
 J1 =        F
 K1 =        7
 L1 =       3F
```

18. These are Fortran 90 intrinsics. They have no user-callable specific functions. For a more detailed explanation of their use, refer to Appendix C, "Fortran 90 compatibility."
19. No user-callable specific functions are available for these argument/result pairs.

Commonly used library routines

Commonly used library subroutines and functions are listed below. Functions are denoted as such; all other listed utilities are subroutines. For more information refer to the *CONVEX Fortran User's Guide*, Chapter 6, "System utilities."

DATE (*buf*)

Returns current date as *dd-mmm-yy*, where *dd* and *yy* are numeric characters and *mmm* is a three-letter abbreviation for the month. *buf* is of type CHAR*9. This subroutine behaves differently in `-cfc` mode; refer to Appendix D, "Cray Fortran compatibility."

IDATE (*iarray*)

Returns current date in *iarray*, a 3-element array of type INTEGER. *iarray(1)* contains the day, *iarray(2)* contains the month as a value between 1 and 12 and *iarray(3)* contains the year as a value 1969.

ERRSNS (*fnum*, *rmssts*, *rmsstv*, *iunit*, *condval*)

Returns information about last runtime error in *fnum*; remaining arguments are not used. All arguments are of type INTEGER*4.

EXIT (*status*)

Ends a process and makes the argument *status* available to the parent process. *status* is of type INTEGER*4.

SECNDS (*x*)

(function) Returns system time minus the value of its argument in seconds. Both *x* and `secnds(x)` are of type *REAL*4*.

TIME (*buf*)

Returns current system time in an ASCII string as *hh:mm:ss*. *buf* is of type CHAR*8.

RAN (*i*)

(function) Returns random integer; similar to ConvexOS utility `rand`, except that `ran` returns type REAL*4.

MVBITS (*m*, *i*, *len*, *n*, *j*)

Transfers *len* bits from position *i* through *i+len*-1 of source location *m* to position *j* through *j+len*-1 of destination location *n*; $0 \le i, j < 32$; $0 \le i+len, j+len < 32$. All arguments are of type INTEGER*4. If the `-cfc`, `-p8`, or `-pd8` option is specified, then $0 \le i, j < 64$ and $0 \le i+len, j+len < 64$.

FORTRAN 66 compatibility

B

The CONVEX Fortran compiler adheres to the American National Standard FORTRAN 77, X3.9-1978, ISO 1539-1980(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 usage must be modified.

Compiling FORTRAN 66 programs

To compile a FORTRAN 66 program, you can modify the program using the following procedure, transforming it into a FORTRAN 77 program, 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. These changes can then be made manually in an editor or automatically through use of a shell script.

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. If you are running the C shell, you can avoid including the `-F66` option in the `fc` command each time you wish to invoke the FORTRAN 66 compiler by using an alias. `alias` is a C shell command with the following form:

```
alias fc 'fc -F66'
```

You can include this format in your .cshrc file, with the $ parameter representing specified files. For more information, see the csh(1) man page.

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 FORTRAN 77 and FORTRAN 66. You must substitute an equivalent statement to include the changes, as shown below.

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)

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.

For example, in FORTRAN 77, the loop

```
DO 20 J=INIT,LAST
```

is not executed if `INIT` is greater than `LAST`, but is executed once in FORTRAN 66.

If this DO statement occurs in a FORTRAN 66 program, its equivalent FORTRAN 77 statement is as follows:

```
DO 20 J=INIT,MAX(INIT,LAST)
```

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 that are used under FORTRAN 66 and VAX Fortran.

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, unless the `-vfc` option is specified or the COVUE shell is used, the OPEN statement BLANK keyword defaults to `BLANK='NULL'` (which means that blanks in numeric fields are ignored). The FORTRAN 66 (and VAX Fortran) interpretation of blanks in numeric input fields is equivalent to `BLANK='ZERO'`.

When a logical unit is opened without an explicit OPEN statement and the `-F66` option or `-vfc` option is specified on the compiler command line, CONVEX Fortran provides a default that is 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.

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.

X descriptor

The FORTRAN 66 implementation of the `X` format edit descriptor writes blanks to the output record and can extend it. 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.

Format code separators

Formats without format code separators are supported.

Fortran 90 compatibility

C

The CONVEX Fortran compiler adheres to the American National Standard programming language FORTRAN 77, X3.9-1978, ISO 1539-1980(E). The default language interpretations are FORTRAN 77 with default CONVEX extensions and features of Fortran 90.

To provide Fortran 90 support, CONVEX Fortran is compatible with certain features of International Fortran Standard, ISO/IEC 1539.1991, which is identical to the ANSI Fortran 90 programming language, ANSI X3.198-1992.

Fortran 90 features available in CONVEX Fortran include:

- `IMPLICIT NONE` statement
- `INCLUDE` statement
- `DO` and `WHILE` loop extensions
- Allocatable arrays
- Automatic arrays
- Array sections
- Vector-valued subscripts
- Array-valued expressions
- Array constructors
- Masked array assignments (`WHERE` statement and construct)
- Certain Fortran 90 array manipulation intrinsic routines

Each of these features is explained in the appropriate section of this manual. This appendix consolidates detailed descriptions of each feature, including usage examples where appropriate.

Array declarations

An array's declaration defines the name of the array within the program unit, the number of dimensions (the array's rank), and the upper and lower bounds of elements in each dimension (its shape).

For multidimensional arrays, separate the dimension declarators with commas. DIMENSION, COMMON, ALLOCATABLE, POINTER, and type statements allow array declarations. For more information, consult Chapter 3, "Arrays and substrings."

For information about declaring arrays of type GATE or BARRIER, refer to Chapter 12, "SPP Series synchronization."

Allocatable arrays

An allocatable array can change shape during program execution and is local to the subroutine in which it is declared. Although an allocatable array keeps the same rank (number of dimensions) throughout the program, storage for it is dynamically allocated on the heap at runtime, so the size of the array's dimensions can change.

When storage allocated to an allocatable array is deallocated (released), the array can be reallocated to occupy an amount of storage which may differ from the previous allocation. For more information, see Chapter 3, "Arrays and substrings."

Automatic arrays

An automatic array is local to a subroutine and contains one or more non-constant dimensions. Memory for an automatic array is allocated on the heap on entry into the subroutine; this storage is released on exit from the subroutine.

Automatic arrays cannot be saved using the SAVE statement, so all information stored in an automatic array is lost on exit from the subroutine in which it is used. For more information, see Chapter 3, "Arrays and substrings."

Array references

CONVEX Fortran allows you to specify sections of arrays to be used as operands or arguments and it supports the use of vector subscripts for referencing multiple, discontinuous elements of an array. Both these methods for referencing array elements are described in this section. For more information about referencing array elements, refer to Chapter 3, "Arrays and Substrings."

Array sections

CONVEX Fortran allows you to use array sections as operands in assignment statements, as arguments to user-written subprograms, and as arguments to many intrinsic functions as specified in Appendix A, "Intrinsics and commonly used library routines." An array section is a piece of an array bounded by specified elements in each dimension. The typical example of a rectangular section of a two-dimensional array is presented further on. An array section is denoted with the following form:

arrname`(`*i*`:`*j*`[:`*step*`][, ...])`

where

arrname

is the array name.

i

is a scalar constant, variable, or expression describing the element at which the section starts for that particular dimension of the array. *i* defaults to the declared lower bound of that dimension of the array.

j

is a scalar constant, variable, or expression describing the element at which the section ends for that particular dimension of the array. *j* defaults to the declared upper bound of that dimension of the array.

step

is a scalar constant, variable, or expression describing the number of elements to step along that particular dimension when selecting elements for the array section. *step* defaults to 1.

Given the defaults for each of these values, array section specifiers such as `X(::1)` and `X(:)` are allowed. These particular examples are equivalent to the entire array, which can also be denoted by `X`.

Note

Array section notation has similar syntax to rank definition, but differs according to usage context.

Following is an example of how a rectangular section of a two-dimensional array is denoted:

```
Y = X(5:10,1:20:2)
```

This example assigns a section of the array X consisting of rows 5 through 10 and odd-numbered columns 1 through 20 to the array Y (a step of 2 starting at column 1 yields odd-numbered columns). This section has rank 2 and shape (6,10). Y must be declared as a two-dimensional array with exactly the same shape and rank as the section. Any time an array section is assigned to another array section, the array sections must have identical shape. Array sections of shape (1, 1)—for example, A(3:3,1:1)—are still considered arrays and therefore are not accepted where a scalar variable is required. In other words, A(3:3,1:1) is not equivalent to A(3,1); the former describes a single element array section and the latter describes a scalar value.

Vector subscripts

Vector subscripts provide a way to reference multiple, discontinuous elements of an array. A vector subscript is a rank-one array of INTEGER values used as a subscript to reference multiple elements of an array. All values in a vector subscript must fall within the dimensions of the referenced array. The following example references four elements of array A.

```
INTEGER K(4)
REAL A(10)
...
K = (/ 1, 7, 3, 4 /)
A(K) = 47.0
```

The above example sets elements 1 to 4 of array K to be 1, 7, 3, and 4, respectively, using an array constructor (which is described later in the "Array constructors" section). By using a vector subscript to reference elements of array A, elements 1, 7, 3, and 4 of A are assigned a value of 47.0.

Array assignments

CONVEX Fortran supports the use of array-valued expressions in assignment statements, the use of array constructors to assign multiple values to elements in an array, and masked array assignments (WHERE statements and constructs). These methods of manipulating array data are discussed in this section.

Array-valued expressions

The ability to use array-valued expressions in assignment statements is supported by CONVEX Fortran. This feature

allows you to assign a value or expression to an entire array or array section with one assignment, using the following form:

arr = *ex*

where *arr* is the name of an array or an array section description and *ex* is an expression.

As with assignment to a scalar variable, mismatched expression types are converted to the type of the argument on the left before assignment.

```
INTEGER IX(10)
REAL X
     .
     .
     .
IX = IX * X
```

Here, each element of IX is multiplied by the REAL variable X and truncated to an INTEGER. The result then replaces the original element in the array.

Fortran 90 array assignments are translated into loops by the compiler. Any optimization options specified at compile time are then applied to the generated loop; for instance, on a C Series machine the loop would be vectorized at optimization levels -O2 and higher.

Array sections can be similarly assigned. Refer to the section "Array sections" in this appendix for more information.

Array constructors

CONVEX Fortran supports Fortran 90 array constructors, which facilitate assigning values to elements of an array. An array constructor may be a scalar expression, an array expression, an implied-DO specification, or a combination of all three forms.

Array constructor assignments use the following form:

arr = (/ *arr_constr_list* /)

where

arr is an array or an array section and *arr_constr_list* is a list of scalar values, array expressions, or implied-DO specifications to be assigned to *arr*. Both *arr_constr_list* and *arr* must be of the same type and shape.

The following example assigns values to elements 1 through 10 of the REAL array COST:

```
COST(1:10) = (/ 9.95, SALE(1:7), 2.75, 4.98 /)
```

Following the assignment, COST(1) is 9.95, COST(2) has the value of SALE(1), COST(3) has the value of SALE(2), and so on.

Likewise, in the following example two array constructors are used, one to specify elements in array P for assignment, and one to specify values for those elements.

```
INTEGER L(4)
REAL P(75)
.
.
.
L = (/2, 36, 7, 9/)
P(L)= (/1.3, 9.2, 12.3, 10.1/)
```

After these assignments, P(2) has a value of 1.3, P(36) is 9.2, P(7) is 12.3, and P(9) is 10.1.

Implied-DO array constructor

The implied-DO form of array constructor has the form

```
( dlist, i = m1, m2 [, m3] )
```

where

dlist

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

i

is the name of an integer variable (the implied-DO variable). The implied-DO variable has the scope of the implied-DO loop. If an array constructor contains more than one implied-DO loop, each implied loop must have its own implied-DO variable.

m1, m2, m3

are integer constant expressions that specify the initial value, terminal value, and increment, respectively, for the implied-DO variable. If you omit *m3*, the increment defaults to 1. The increment must be positive.

These constants can contain implied-DO variables of other implied-DO lists.

The following assignment uses the implied-DO array constructor:

```
REAL A(10)
A = (/ (SQRT(REAL(I)), I = 0, 27, 3) /)
```

The above statement assigns the square root of I to all ten elements of array A. Because the SQRT intrinsic accepts only REAL*4 values, the integer I is converted to this type using the REAL intrinsic. A(1) is assigned SQRT(0.0), A(2) is assigned SQRT(3.0), A(3) is SQRT(6.0), and so on.

The following code is equivalent to the previous implied-DO array constructor:

```
I = 1
DO J = 0, 27, 3
   A(I) = SQRT(REAL(J))
   I = I + 1
END DO
```

Masked array assignment

CONVEX Fortran allows assignment of values to an array under a mask specified via the WHERE statement or WHERE construct. The WHERE statement evaluates a logical expression to determine to which elements the assignment is being applied. The WHERE construct works similarly, but is terminated with the ENDWHERE statement. It can contain several assignments and an ELSEWHERE statement, which allows alternate assignments to be applied to the complement of the mask-expression.

The WHERE statement has the following form:

WHERE(*mask-expr*) *assignment-stmt*

The WHERE construct has the following form:

```
WHERE (mask-expr)
   [assignment-stmt]
   [...]
[ELSEWHERE
   [assignment-stmt]
   [...]
ENDWHERE
```

where

mask-expr

is a logical expression of the same shape as the array(s) being manipulated in the *assignment-stmt(s)*.

assignment-stmt

is an array assignment. The array must be the same shape as the array in *mask-expr*. If a function is used here, any array arguments it takes must also be of the same shape.

On execution, the *mask-expr* is evaluated. Any following assignments are executed only on array elements corresponding to those elements for which *mask-expr* evaluated to .TRUE. If an ELSEWHERE statement is present, its assignments are applied to array elements corresponding to those elements for which *mask-expr* evaluated to .FALSE.

In the WHERE construct, *mask-expr* is evaluated once at the beginning of the construct and the result stored for use in every *assignment-stmt*. Execution of the construct then proceeds as if each *assignment-stmt* was part of a WHERE statement using the original evaluation of *mask-expr*, as demonstrated below.

```
WHERE (mask-expr)
   assignment-stmt1
   assignment-stmt2
ELSEWHERE
   assignment-stmt3
ENDWHERE
```

This WHERE construct is equivalent to the following series of WHERE statements.

```
WHERE(mask-expr) assignment-stmt1
WHERE(mask-expr) assignment-stmt2
WHERE(.NOT.(mask-expr)) assignment-stmt3
```

Each WHERE statement is then further decomposed into a DO-loop by the compiler to facilitate the actual assignment of values into the array elements. It is important to note that:

- *mask-expr* is evaluated only once before entering the body of the WHERE and does not change over the course of the construct, even if the array on which it is based is changed in the WHERE body.
- Each *assignment-stmt* is executed in full independently of the other *assignment-stmts*. This means that if *assignment-stmt* contains an array assignment, the compiler may generate a DO-loop specifically for that assignment. Do not assume that multiple *assignment-stmts* that involve the same array section will execute sequentially as if part of a single DO-loop.
- The compiler-generated DO-loops are subject to the same optimizations as hand-written DO-loops.

Examples of the WHERE statement and construct follow.

Example 1:

```
REAL DATA(1000),OFFSET(10000),ADJUSTED(1000),LIMIT
LOGICAL NORMAL(1000)
LIMIT = 130.0
      .
      .
      .
WHERE(DATA .LE. LIMIT)  NORMAL = .TRUE.
```

In this example, all elements of the logical array NORMAL that have the same index as elements in the real array DATA that are less than or equal to LIMIT are set to .TRUE.

The following example assumes the same variable declarations as Example 1.

Example 2:

```
WHERE(DATA .GT. LIMIT)
  ADJUSTED = FIX(DATA)
  NORMAL = .FALSE.
ELSEWHERE
  ADJUSTED = 0.0
  NORMAL = .TRUE.
ENDWHERE
```

In this example, elements of ADJUSTED corresponding to elements of DATA greater than 130.0 are set to FIX(DATA), and all other elements of ADJUSTED are set to 0.0. Similarly, all elements of NORMAL corresponding to DATA elements greater than 130.0 are set to .FALSE. and all other elements of NORMAL are set to .TRUE.

Array sections and subscript expressions can be used with the WHERE construct to change the correspondence of elements between arrays. The following example assumes the same variable declarations as Example 1.

Example 3:

```
WHERE(DATA .LE. LIMIT)
  OFFSET(ISET:ISET+999) = DATA
ELSEWHERE
  OFFSET(ISET:ISET+999) = 0.0
ENDWHERE
```

In this example, elements of OFFSET starting at the value ISET and going through ISET+999 are set to correspond to elements of DATA where the DATA elements are less than or equal to LIMIT, and are set to 0.0 where the DATA elements are greater than LIMIT.

Note

It is possible to assign values to the same array element in both the .TRUE. and .FALSE. branches of the WHERE construct when using array sections or subscript expressions that differ in each branch of the construct. This action can inhibit vectorization and parallelization of the WHERE construct, and should therefore be avoided.

Remember, the WHERE construct differs from the block-IF in that both clauses can and often do execute. It is therefore possible for assignments that execute in the ELSEWHERE clause of the WHERE construct to change values assigned in the WHERE clause when array section notation is used. The following example, which illustrates this, assumes the same variable declarations as Example 1.

Example 4:

```
WHERE(DATA .LE. LIMIT)
  OFFSET(ISET:ISET+999:2) = DATA(1:1000:2)
ELSEWHERE
  OFFSET(ISET+1:ISET+1000:2) =
^   OFFSET(ISET:ISET+999:2)
ENDWHERE
```

In this example, every other element from OFFSET(ISET) through OFFSET(ISET+999) is set in the WHERE clause. Then in the ELSEWHERE clause, because of the sectioning notation used, each value assigned in the WHERE clause is copied into the element immediately following it in OFFSET. The end result is that OFFSET(ISET) = OFFSET(ISET+1) = DATA(1), OFFSET(ISET+2) = OFFSET(ISET+3) = DATA(3) and so on for every other element of OFFSET.

Fortran 90 array manipulation intrinsics

CONVEX Fortran includes a subset of Fortran 90 array manipulation intrinsic routines. In CONVEX Fortran, Fortran 90 intrinsics cannot be nested. These intrinsics are explained in the following subsections.

Many Fortran 90 intrinsics provide for optional arguments. The ANSI Fortran 90 standard allows the *keyword = argument* syntax in the argument list so that these optional arguments can be skipped. CONVEX Fortran does not support this syntax. In CONVEX Fortran, if you want to omit an optional argument that is not the final argument in the argument list, you must supply a 0 in its place in the argument list. Optional arguments that fall at the end of the argument list do not require a place holder; they can be left out of the list.

Vector and matrix multiply functions

These routines are used to multiply vectors by vectors, matrices by matrices and vectors by matrices. A *vector* is defined as an

array of rank one. A *matrix* is defined as an array of rank greater than or equal to one.

DOT_PRODUCT
DOT_PRODUCT performs dot-product multiplication of two numeric or logical vectors. It has the following form:

```
DOT_PRODUCT(vecta, vectb)
```

where

vecta

must be an array valued vector of numeric type (INTEGER, REAL or COMPLEX) or of type LOGICAL.

vectb

must be an array valued vector of numeric type if *vecta* is of numeric type or of type LOGICAL if *vecta* is of type LOGICAL. *vectb* must be the same size as *vecta*.

DOT_PRODUCT returns a scalar value. For numeric arguments, the type of this value is the same as the type of (*vecta* × *vectb*); for logical arguments, the result is the same as the type of (*vecta* .AND. *vectb*).

MATMUL
MATMUL performs matrix multiplication of two numeric or logical matrices. It has the following form:

```
MATMUL(mata, matb)
```

where

mata

must be an array valued rank one or rank two matrix of numeric type or type LOGICAL.

matb

must be an array valued rank one or rank two matrix of numeric type if *mata* is of numeric type or of type LOGICAL if *matb* is of type LOGICAL. If *mata* has rank one, *matb* must have rank two; if *matb* has rank one, *mata* must have rank two. The size of the first (or only) dimension of *matb* must equal the size of the last (or only) dimension of *mata*.

MATMUL returns an array with the same type as that of (*mata* × *matb*); for logical arguments, the result is the same as the type of

(*mata* .AND. *matb*). The shape of the array is determined by the arguments:

- If *mata* has shape (*n*,*m*) and *matb* has shape (*m*,*k*), the result has shape (*n*,*k*).
- If *mata* has shape (*m*) and *matb* has shape (*m*,*k*), the result has shape (*k*).
- If *mata* has shape (*n*,*m*) and *matb* has shape (*m*), the result has shape (*n*).

Reduction functions

These functions perform various reduction operations on arrays. All reduction functions have at least one argument (an array) and can have one or two additional optional arguments.

In CONVEX Fortran, when a function with two optional arguments is called and the middle argument is to be omitted, the middle argument must be given a value of 0 as a place holder.

ALL

ALL determines whether all values of an array along an optional dimension are true. ALL has the following form:

```
ALL(mask[, dim])
```

where

mask

an array of type LOGICAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \le dim \le n$, where n is the rank of *mask*. If *dim* is omitted, ALL is applied to the entire array and yields a scalar result.

ALL always yields a result of type LOGICAL. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *mask*. In other words, the shape is identical to the shape of *mask* except it is missing the dimension specified in *dim*.

ANY

ANY determines whether any value in an array along an optional dimension is true. ANY has the following form:

```
ANY(mask[, dim])
```

where

mask

an array of type LOGICAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*. If *dim* is omitted, ANY is applied to the entire array and yields a scalar result.

ANY always yields a result of type LOGICAL. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *mask*.

COUNT

COUNT counts the true elements in an array along an optional dimension. COUNT has the following form:

```
COUNT(mask[, dim])
```

where

mask

an array of type LOGICAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*. If *dim* is omitted, COUNT is applied to the entire array and yields a scalar result.

COUNT always yields a result of type INTEGER. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *mask*.

MAXVAL

MAXVAL returns the maximum value of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. MAXVAL has the following form:

```
MAXVAL(array[, dim[, mask]])
```

where

array

an array of type INTEGER or REAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \le dim \le n$, where n is the rank of *mask*.

mask (optional)

must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX Fortran, if you want to omit *dim* and supply *mask*, you must supply a 0 in *dim's* position.

MAXVAL yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

MINVAL

MINVAL returns the minimum value of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. MINVAL has the following form:

```
MINVAL(array[, dim[, mask]])
```

where

array

an array of type INTEGER or REAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \le dim \le n$, where n is the rank of *mask*.

mask (optional)

must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX Fortran, if you want to omit *dim* and supply *mask*, you must supply a 0 in *dim's* position.

MINVAL yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

PRODUCT

PRODUCT returns the product of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. PRODUCT has the following form:

```
PRODUCT(array[, dim[, mask]])
```

where

array

an array of type INTEGER, REAL or COMPLEX.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*.

mask (optional)

must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX Fortran, if you want to omit *dim* and supply *mask*, you must supply a 0 in *dim's* position.

PRODUCT yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

SUM

SUM returns the sum of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. SUM has the following form:

```
SUM(array[, dim[, mask]])
```

where

array

is an array of type INTEGER or REAL.

dim (optional)

must be a constant scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*.

mask (optional)

must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX Fortran, if you want to omit *dim* and specify a *mask*, you must supply a 0 in *dim's* position.

SUM yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

Construction functions

The functions listed in this section are used to construct arrays of all types.

MERGE

MERGE constructs an array by merging two source arrays under a mask. MERGE has the following form:

```
MERGE(tsource, fsource, mask)
```

where

tsource

is an array or scalar of any type. The elements in this array are masked by the .TRUE. elements of *mask*. If *tsource* is a scalar and *fsource* is an array, the value of *tsource* is broadcast to cover the shape of *fsource*.

fsource

is an array or scalar of the same type as *tsource*. In CONVEX Fortran, *fsource* must be conformable with *tsource*. The elements in this array are masked by the .FALSE. elements of *mask*. If *fsource* is a scalar and *tsource* is an array, the value of *fsource* is broadcast to cover the shape of *tsource*.

mask

is an array of type LOGICAL that represents a mask under which *tsource* and *fsource* are merged. *mask* must be conformable with *tsource*.

MERGE returns an array of the same type as *tsource*. The resulting array consists of elements of *tsource* that correspond to the .TRUE. elements of *mask* and elements of *fsource* that correspond to the other elements of *mask*. If *tsource* is an array, the result is of the same shape; if *tsource* is a scalar but *mask* is an array, the result is of the same shape as *mask*. If all three arguments are scalars, the result is *tsource* if *mask* is .TRUE., *fsource* otherwise.

PACK

Packs the elements of an array into an array of rank one under control of a mask. PACK has the following form:

```
PACK(array, mask, vector)
```

where

array

is an array of any type.

mask

must be an array or scalar of type LOGICAL and must be conformable with *array*. The resulting array will contain only those elements of *array* that correspond to .TRUE. elements of *mask*.

vector

is a rank one array of the same type as *array*, containing at least as many elements as there are .TRUE. elements in *mask*. If *mask* is a scalar, *vector* must contain at least as many elements as *array*.

PACK returns a rank one array with the same type as *array*. The resulting array is the same size as *vector*. If *vector* contains more elements than those masked out of *array*, the masked *array* elements fill *vector* consecutively from its beginning and any leftover *vector* elements are unchanged. Note that while *vector* is an optional argument according to the ANSI Fortran 90 standard, it is required in the CONVEX Fortran implementation of PACK.

SPREAD

SPREAD replicates an array into a specified new dimension a specified number of times. SPREAD has the following form:

```
SPREAD(source, dim, ncopies)
```

where

source

is an array or scalar of any type. It must have a rank of less than seven.

dim

must be a constant scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*.

ncopies

must be a scalar of type INTEGER. Specifies the number of copies of *source* to be made.

SPREAD returns an array of the same type as *source* with rank $n+1$, where n is the rank of source. If *source* is scalar, the shape of the result is (MAX(0,*ncopies*); if *source* is an array with shape $(d_1, d_2, ..., d_n)$, the shape of the result is $(d_1, d_2, ..., d_{dim-1}, MAX(0, ncopies), d_{dim}, ..., d_n)$

UNPACK

Unpacks a rank one array into an array under control of a mask. UNPACK has the following form:

```
UNPACK(vector, mask, field)
```

where

vector

is a rank one array of any type. It must have at least as many elements as there are .TRUE. elements in *mask*. *vector* contains the array to be unpacked.

mask

must be an array of type LOGICAL. *mask* provides the shape that *vector* will be unpacked into.

field

must be of the same type as *vector* and must be conformable with *mask*; scalar values are acceptable.

UNPACK returns an array of the same type as *vector* having the shape of *mask*. The positions in the result that correspond to the positions of the .TRUE. elements of *mask* are filled, in array element order, with the values from *vector*. All other elements of the result contain the value of *field* if *field* is a scalar, or corresponding elements of *field* if *field* is an array.

Manipulation functions

These functions are used to manipulate arrays.

CSHIFT

CSHIFT performs a circular shift on the positions of elements parallel to a specified dimension of the array; elements shifted off one end reappear at the other end.

CSHIFT has the following form:

```
CSHIFT(array, shift, dim)
```

where

array

is an array of any type.

shift

must be of type integer and must be scalar if *array* has rank one; otherwise, *shift* must be scalar or of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

dim (optional)

must be a scalar of type INTEGER with a value in the range 1 *dim* *n*, where *n* is the rank of *array*. If *dim* is omitted, it is as if it were present with the value 1.

CSHIFT returns an array of the same type and shape as *array*.

EOSHIFT

EOSHIFT performs an end-off shift on an array expression of rank one or on each complete rank-one section along a given dimension of an array expression of rank two or greater. Elements are shifted off at one end and copies of a boundary value are shifted into vacant elements at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.

EOSHIFT has the following form:

```
EOSHIFT(array, shift, boundary, dim)
```

where

array

is an array of any type.

shift

must be of type integer and must be scalar if *array* has rank one; otherwise, *shift* must be scalar or of rank *n-1* and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

boundary (optional)

must be of the same type and type parameters as *array* and must be scalar if *array* has rank one; otherwise, it must be either scalar or of rank *n -1* and of shape $(d_1, d_2, ..., d_n)$. *boundary* may be omitted for the data types shown below and, in this case, it is as if it were present with the scalar value shown.

Type of *array*	**Value of *boundary***
`INTEGER`	`0`
`REAL`	`0.0`
`COMPLEX`	`(0.0,0.0)`
`LOGICAL`	`.FALSE.`
`CHARACTER*(`*len*`)`	*len* blanks

dim (optional)

must be a scalar of type `INTEGER` with a value in the range 1 *dim* *n*, where *n* is the rank of *array*. If *dim* is omitted, it is as if it were present with the value 1.

`EOSHIFT` returns an array of the same type and shape as *array*.

`TRANSPOSE`

`TRANSPOSE` returns the transpose of an array of rank two. `TRANSPOSE` has the following form:

```
TRANSPOSE(matrix)
```

where

matrix

is a matrix of any type. Must be of rank two.

`TRANSPOSE` returns a rank two array of the same type as *matrix* and with shape (m, n) where (n, m) is the shape of *matrix*. In other words, the rows and columns of *matrix* are swapped.

Location functions

These functions are used to locate the maximum and minimum values of the array.

MAXLOC
MAXLOC locates the maximum value in an array (along an optional mask if present). MAXLOC has the following form:

```
MAXLOC(array[, mask])
```

where

array

must be an array of type INTEGER or REAL. *array* must not be a scalar.

mask (optional)

must be a type LOGICAL array that is conformable with *array*.

MAXLOC returns a rank one array of type INTEGER and of size equal to the rank of *array*. The elements of the result are the subscripts of *array* which locate the maximum value of *array*. If this value occurs in more than one element of *array*, the location of the first occurrence is returned. If *mask* is present, the element returned is the maximum of those elements of *array* that correspond to true elements of *mask*.

MINLOC
MINLOC locates the minimum value in an array (along an optional mask if present). MINLOC has the following form:

```
MINLOC(array[, mask])
```

where

array

must be an array of type INTEGER or REAL. *array* must not be a scalar.

mask (optional)

must be a type LOGICAL array that is conformable with *array*.

MINLOC returns a rank one array of type INTEGER and of size equal to the rank of *array*. The elements of the result are the subscripts of *array* which locate the minimum value of *array*. If this value occurs in more than one element of *array*, the location of the first occurrence is returned. If *mask* is present, the element

returned is the minimum of those elements of *array* that correspond to true elements of *mask*.

Cray Fortran compatibility

D

To facilitate porting code written for the Cray Fortran compiler, CONVEX Fortran supports the `-cfc` compiler option, which provides compatibility with Cray Fortran as described in this appendix.

By default all Cray Fortran extensions are disabled.

Supported Cray features

When the `-cfc` compiler options is specified, CONVEX Fortran provides compatibility with Cray Fortran in the following ways:

- Intrinsics that work with CONVEX default integer and single-precision types work with Cray default types (8-byte quantities).
- Double-precision (`REAL*16`) data types are supported on CONVEX SPP Series machines and in native mode on CONVEX C Series machines. Constants written with a `D` in the exponent or objects declared double-precision are treated as 16-byte objects.
- Logical constants `.T.` and `.F.` are supported.
- The Cray `POINTER` statement is supported. For compatibility with Cray pointer arithmetic, specify both the `-cfc` and `-cfcwpa` flags.
- Cray automatic arrays are supported.
- A look-alike implementation of the Cray `BUFFERIN`, `BUFFEROUT` features and accompanying routines is available.
- Read and write access to unformatted Cray data files is provided.
- Cray `TASK COMMON` blocks are supported.
- Cray Boolean octal constants and left-justified Cray Hollerith constants are accepted.

- Constants are stored in the Cray default `INTEGER` or `REAL` types. See the "Cray data types" section for more information.
- Constants written in single-precision exponential form (such as `1.23E4`) are stored in `REAL*8` format.
- `LOGICAL*2` and `*4`, `INTEGER*4`, and `REAL*4` are treated as `LOGICAL*8`, `INTEGER*8`, and `REAL*8`.

Unsupported Cray features

Only those Cray-specific intrinsics enumerated in the "Supported Cray intrinsics" section of this appendix are supported. Cray-specific compiler directives are not supported.

Also, when a program uses the `LOC` function, Cray Fortran returns a word address; CONVEX Fortran returns a byte address.

Cray data types

The default data type lengths used by Cray Fortran differ from those used by CONVEX Fortran. The `-cfc` option instructs the compiler to use Cray Fortran data type lengths rather than CONVEX Fortran data type lengths.

Cray Fortran's default data types are as follows:

Type	Default length
Integer	`INTEGER*8`
Real	`REAL*8`
Complex	`COMPLEX*16`
Logical	`LOGICAL*8`
Double precision	`REAL*16`

The CONVEX/VAX representation of `REAL*16` data gives more precision (113 bits) than the Cray, IBM, and IEEE specifications. This greater accuracy makes the CONVEX `REAL*16` software much slower (possibly 40 times slower) than Cray double-precision software because of the greater number of intermediate results that must be generated and saved internally. On C Series machines, the `REAL*16` data type is available only in native mode.

Cray POINTER support

The Cray POINTER statement is supported. However, normally in CONVEX Fortran pointer arithmetic computes byte addresses rather than word addresses. Due to this difference you must either multiply your pointer offsets by eight or use the -cfcwpa compiler option for word addresses. Using the -cfcwpa option requires specifying the -cfc compiler option.

For pointers that are specifically declared using the POINTER statement, addition or subtraction of pointers will compute word addresses under the -cfcwpa option. The compiler will issue a warning when it encounters other arithmetic operations on pointers. If you do not specify the -cfcwpa option and the compiler encounters an explicitly declared pointer in an arithmetic context, a warning message is issued; this option must be specified in order to perform Cray pointer arithmetic.

Note

The -cfcwpa option recognizes only pointers that have been specifically declared using the POINTER statement. Byte arithmetic is performed on pointers that are not explicitly declared.

For more information on the -cfcwpa option refer to Chapter 1, "Compiling programs," of the *Fortran User's Guide*.

The syntax for the POINTER statement is

```
POINTER(p,s)
```

where

p is a pointer, and *s* is the name of a local variable or array. In Cray terminology, *s* is the pointee. *s* cannot be associated with any other known piece of named and referenced storage except through assignments to *p* or by associating two or more pointees with one pointer.

More than one pointer should *not* point to the same storage, but this optimization requirement is not enforced by the compiler.

Pointees are assumed *not* to overlap. This means that a value stored indirectly through one pointee must not be accessed via another pointee even when both pointees are associated with the same pointer.

Debugging code containing Cray pointers

csd, the CONVEX Symbolic Debugger, is an optional product. It cannot access Cray pointers. However, you can dump the contents of the pointer in csd; this gives you the raw memory

address of the pointee. You then can view the contents of this address in csd.

Example:

```
PROGRAM POINT
REAL A(10)
POINTER (IPA, A)
CALL HPALLOC (IPA, 10, IA, IR)
DO I = 1, 10
  A(I) = I
END DO
PRINT *, A
END
```

To examine the contents of A from within csd, first get its address (the value of IPA):

```
(csd) p ipa
2148233220
```

Now you can examine the memory locations beginning at 2148233220 to see the contents of the array A:

```
(csd) 2148233220,10?F
800b9004:  1.000000 2.000000
800b9014:  3.000000 4.000000
800b9024:  5.000000 6.000000
800b9034:  7.000000 8.000000
800b9044:  9.000000 10.000000
```

Refer to the *CONVEX Consultant User's Guide* for more information on the use of csd.

Cray automatic arrays

CONVEX Fortran supports Cray automatic arrays under the -cfc option as well as the -f90 compiler option (-f90 is enabled by default). Any array declared local to a subroutine with one or more non-constant dimensions is considered an automatic array. Memory for automatic arrays is dynamically allocated on the stack on entry into the subroutine based on a variable dimension value passed into the subroutine. This stack space is freed on exit from the subroutine; automatic arrays cannot be saved using the SAVE statement.

Automatic arrays are described in detail in Chapter 3, "Arrays and substrings," and Appendix C, "Fortran 90 compatibility."

Cray BUFFERIN, BUFFEROUT support

CONVEX provides a look-alike implementation of the Cray BUFFERIN, BUFFEROUT feature and its accompanying routines. This feature is provided mainly to aid in porting existing Cray code; it is not likely to provide noticeably superior performance compared to conventional CONVEX I/O methods. BUFFEROUT will always be slightly slower than unformatted fixed record length I/O.

Related statements and routines

The following statements are available for use with BUFFERIN, BUFFEROUT:

```
BUFFERIN (unit, mode) (beginLoc, endLoc)

BUFFEROUT (unit, mode) (beginLoc, endLoc)
```

These library routines are also available: UNIT(*unit*), LENGTH(*unit*), GETPOS(*unit*) and SETPOS(*unit*, *pos*). These routines reside in `libcfc.a`, which is implicitly loaded when linking with the `-cfc` compiler option.

Note

If present, data format conversions specified in the OPEN statement do not affect data read or written with BUFFERIN or BUFFEROUT statements.

Restrictions

Note the following restrictions:

- The SETPOS function cannot be used with magnetic tape; doing so produces an error message.
- Using BUFFERIN and BUFFEROUT with data types less than eight bytes produces a fatal compiler error.
- Other Fortran I/O statements (READ, WRITE, PRINT, ACCEPT, TYPE) cannot be used on the same unit as BUFFERIN, BUFFEROUT.
- BACKSPACE does not work with BUFFERIN, BUFFEROUT files.
- The SPP Series unformatted sequential access test (described in the "FORM keyword" section of Chapter 9) should *not* be enabled for already existing files that are created with BUFFEROUT statements and are opened as unformatted sequential access files.

Cray unformatted file support

CONVEX Fortran is capable of reading and writing unformatted Cray data files through use of the `FORM = UNFORMATTED/CRAY` and `FORM = UNFORMATTED/CRAYUB` keyword definitions in the `OPEN` statement. Formatted files are supported regardless of file structure or access mode; unformatted files are partially supported depending on file structure and access mode. Table 36 shows the degree of support provided given various combinations of file structures and access modes.

Table 36 Unformatted Cray files readable by CONVEX Fortran

Cray file structure	Access mode	
	Sequential	**Direct**
Unblocked (pure)	Does not comply with the ANSI Fortran 77 standard; no record boundaries inherent in file; `BACKSPACE` not permitted; use `FORM=UNFORMATTED/CRAYUB` in `OPEN` statement.	Must specify `FORM=UNFORMATTED/CRAY` in `OPEN` statement unless data type conversions were performed during Cray creation of the file.*
Blocked	Must run `fcUnblock`† utility on these files to convert them into a form readable by CONVEX Fortran. Then open the file with `FORM=UNFORMATTED/CRAY`.*	Cray Fortran does not permit.

* These file structures are the default in Cray Fortran.
† For more information on the `fcUnblock` utility, see the fcUnblock (1F) man page.

Supported Cray library routines

The following Cray library routines are available when using the `-cfc` compiler option:

`cft$bool`	`cft$dprod`	`clock`	`date`	`dump`
`etime`	`findch`	`gather`	`hpalloc`	`hpcheck`
`hpclmove`	`hpdeallc`	`hpdump`	`hpnewlen`	`hpshrink`
`iceil`	`igtbyt`	`ihplen`	`ihpstat`	`ilsum`
`int24`	`jdate`	`komstr`	`lint`	`mvc`
`pack`	`putbyt`	`rbn`	`rnb`	`scopy`
`sdot`	`second`	`ssum`	`strmov`	`timef`
`tr`	`unpack`			

Additionally, most `libU77.a` routines can be accessed in Cray mode. The exceptions are: `dbesj0`, `dbesj1`, `dbesjn`, `dbesy0`, `dbesy1`, `dbesyn`, `derf`, and `derfc`. For more information about `libU77.a` routines, refer to the intro.3f man page.

Note

A Cray mode call to `date` will access the Cray specific `date` routine, which returns the date in a format different from that returned from the `libU77.a date` routine. Similarly, a Cray mode call to `rand` accesses the Cray routine and returns a different sequence of numbers than the corresponding call to the `libU77.a rand` routine.

Supported Cray intrinsics

The following Cray intrinsics are available under CONVEX Fortran's `-cfc` option:

`and`	`compl`	`cot`	`csmg`	`cvmgm`
`cvmgn`	`cvmgp`	`cvmgt`	`cvmgz`	`dcot`
`eqv`	`leadz`	`loc`	`mask`	`movbit`
`neqv`	`or`	`popcnt`	`poparr`	`ranf`
`ranget`	`ranset`	`shift`	`shiftl`	`shiftr`
`xor`				

Additionally, all Military Standard (MIL-STD-1753) routines are supported from Cray mode.

Cray TASK COMMON support

CONVEX Fortran supports Cray TASK COMMON blocks under the -cfc option. A program should already be running multiple threads before calling a subroutine that contains a TASK COMMON block.

On C Series machines, variables in a TASK COMMON block are provided thread-local storage. On SPP Series machines, variables in a TASK COMMON block are stored in a thread-private COMMON block.

The TASK COMMON statement creates these blocks and has the following form:

```
TASK COMMON /cbn/nlist[ ,/cbn/nlist] . . .
```

where

cbn

is a symbolic name for a TASK COMMON block. Unnamed TASK COMMON blocks are not allowed.

nlist

is a list of variable names, array names, and array declarators. These variables cannot appear in a DATA statement, but otherwise can be used like any variables in COMMON storage.

All occurrences of the TASK COMMON block must be declared TASK COMMON; a common block cannot be declared both COMMON and TASK COMMON. TASK COMMON blocks can only be declared in functions, subprograms and BLOCK DATA subprograms.

Cray Boolean octal constant support

When -cfc is specified, CONVEX Fortran supports Cray Boolean octal constants. Cray Boolean octal constants consist of one or more octal digits followed by the letter B. A Boolean octal constant has the following form:

```
cc...cB
```

where *c* represents an octal digit. There can be up to 22 octal digits in an octal constant. An octal digit can range from 0 to 7.

Like all Cray constants, octal constants occupy 8 bytes (equivalent to one 64-bit Cray word) in memory. It follows that octal constants can be 22 octal digits long, but, because 22 octal digits represent 66 bits in memory, the leftmost octal digit must be either 0 or 1. Octal constants are right justified and zero-filled

to the left. An octal constant that occupies all 22 digits cannot have a value greater than 1 as its leftmost octal digit. Blanks within an octal constant are ignored.

Examples:

Valid	Invalid	Reason
`765B`	`835B`	8 not in range 0 to 7
`1126752354176524376524B`	`3126752354176524376524B`	Leftmost digit > 1

Cray Hollerith constants

When `-cfc` is supplied on the `fc` command line, CONVEX Fortran supports left-justified Cray Hollerith constants of the following form:

*n*L*cc...c*

or

'*cc...c*'*L*

where

n

specifies the number of the characters in the constant (including spaces and tabs). The value of *n* must be an unsigned positive integer greater than zero.

c

is a printable ASCII character.

Cray Hollerith constants occupy 8 bytes (equivalent to one 64-bit Cray word) in memory. Each byte contains one ASCII character code, and Hollerith constants using this form cannot exceed eight characters in length. This form left-justifies the constant in memory and zero-fills its 8-byte storage space to the right.

Example:

Valid	Invalid	Reason
`4LHelp`	`4L'Help'`	No quote in this form.
`'foo'L`	`3'foo'L`	Number of characters specifier not allowed in this form.

VAX Fortran compatibility

E

This appendix describes compatibility between VAX Fortran and CONVEX Fortran. To facilitate porting code written for the VAX Fortran compiler, CONVEX Fortran provides the `-vfc` option to support certain VAX features as described in this chapter.

By default, all VAX Fortran extensions are disabled.

Supported features

CONVEX Fortran supports the following VAX Fortran features when the `-vfc` coption is specified on the `fc` command line:

- `REAL*16` data type (on CONVEX SPP Series machines and in native mode on CONVEX C Series machines)
- VAX `INCLUDE` statement
- 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
- VAX Fortran `RECORD`s, including the `RECORD`, `STRUCTURE`, `UNION` and `MAP` statements.
- Hollerith constants where a `CHARACTER` value is expected
- The `RECL=` specifier used in the `OPEN` and `INQUIRE` statements. This specifier returns the number of VAX words rather than bytes for unformatted files. (This is not true for programs compiled and loaded separately unless VAX compatibility is enabled for the load phase.)
- Default file names in the form `FOR0`*nn*`.DAT`, where the number *nn* corresponds to unit number *nn*
- The `OPEN` statement `BLANK` keyword defaults to `BLANK = 'ZERO'`.

The organization and structure of VAX Fortran records is discussed briefly at the end of this appendix.

Unsupported features

CONVEX Fortran does *not* support the following VAX Fortran features:

- `%DESCR`
- RMS calls
- The `VOLATILE` statement
- The *zcccc...c* form of hexadecimal constants
- Interactive display of `NAMELIST` group and values or end-of-line comments (`!`) in the `NAMELIST` input data
- Extra parentheses in `WRITE` statements (allowed in VAX Fortran)
- VMS file names
- Byte ordering with respect to passing characters and parameters
- Logical values. On CONVEX C Series computers, a logical value is true if all the bits are 1. On VAX and on CONVEX SPP Series, it is true in a test if the low-order bit is 1, for example, `IF(A)` is equivalent to `IF(A .AND. 1)`.
- Numerical differences. The accuracy of CONVEX floating-point representation and the rounding method used cause these differences. Refer to the *CONVEX Architecture Reference* for further information.
- Automatic conversion of `REAL*8` (in caller) to `REAL*4` (in called subroutine) across subroutine calls
- Calling a function as a subroutine
- Radix 50 constants
- The variable on the left side of a character assignment statement appearing on the right side (VAX Fortran extension)
- `MOD` function defined for a zero denominator
- 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, which is an ANSI standard)

- DO statements of the form:

```
DO 714 J=1, 100 WHILE (Q.NE.Z)
```

- VMS path names. The FILE name you specify when using the INQUIRE statement under the COVUE shell must be the absolute UNIX path name.
- The FILE keyword in the OPEN statement. The keyword must specify the name of the file to open as a character expression; numeric variables, arrays or array elements containing the file name cannot be substituted.

CONVEX Fortran does not support these 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; ORGANIZATION; and SEGMENTED for RECORDTYPE)
- 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.

VAX and CONVEX versions of the STOP message differ as follows:

Statement	CONVEX message	VAX message
STOP	STOP:	Fortran STOP
STOP 4	STOP: 4	4
STOP 'here'	STOP: here	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.

Miscellaneous differences

The following miscellaneous differences exist between VAX Fortran and CONVEX Fortran:

- Unit numbers in VAX Fortran range from 0 to 99 and in CONVEX Fortran from 0 to 255.
- VAX Fortran supports the use of Hollerith and apostrophe edit descriptors during formatted input. CONVEX Fortran allows Hollerith descriptors 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.
- In the `ASSIGN` statement, `ASSIGN s TO i`, where *s* is the label of an executable statement in the current program unit and *i* is an integer variable, VAX Fortran sets the integer variable to 1 at initial execution of the `ASSIGN` statement; CONVEX Fortran sets it to 0.
- 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 CONVEX Fortran, the `READ` statement ends at once and the remaining variables in the *iolist* are unchanged.
- VAX Fortran correctly handles `REAL*16` constants that do not contain `Q` in the exponent. CONVEX Fortran does not; if `Q` is not specified, the constant is considered to be `REAL*4`. On CONVEX C Series machines `REAL*16` is available in native mode only.
- CONVEX Fortran follows the ANSI standard in that any function referenced directly or indirectly in an I/O statement cannot contain another I/O statement. VAX Fortran may not always follow this standard.
- Before VAX binary files can be read by CONVEX Fortran, the `cvbin` utility must be used to convert them. Refer to the cvbin(1) man page or to the *CONVEX COVUEbinary User's Guide* for more information.

VAX Fortran records

A VAX Fortran record is a derived data type containing one or more fields. Fields within a record are defined by a structure declaration.

Structure declaration

A structure declaration defines field names, types of data within fields, and order and alignment of fields within a record.

A structure declaration is bounded by STRUCTURE and END STRUCTURE statements and has 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 that contains the name of a previously declared structure. The RECORD statement has the following form:

```
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.

Records must be read and written using unformatted I/O. For example:

```
C DEFINITION OF THE NAME STRUCTURE
        STRUCTURE / NAME /
           CHARACTER*5 LAST
           CHARACTER*5 FIRST
           CHARACTER*1 INITIAL
        END STRUCTURE
```

```
C DEFINITION OF THE DATE STRUCTURE
      STRUCTURE /DATE/
        CHARACTER*2 MONTH
        CHARACTER*2 DATE
        CHARACTER*2 YEAR
      END STRUCTURE

C DEFINITION OF THE PERSON STRUCTURE
      STRUCTURE / PERSON /
        RECORD / NAME / NAME
        LOGICAL*1 SEX
        RECORD / DATE / BIRTH
      END STRUCTURE

C SET UP ARRAY OF EMPLOYEES TO BE HANDLED
      RECORD / PERSON / EMPLOYEES(3)

C READ EMPLOYEE DATA FROM UNIT 2
      DO I = 1,3
        READ(2) EMPLOYEES(I)
      ENDDO
```

Field declaration

A field declaration can be any combination of the following:

- A typed data declaration
- A substructure declaration
- A union declaration

A typed data declaration is the same as a normal Fortran type statement. Note that a typed data declaration within a structure only declares a field for the structure and does not affect any variables of the same name elsewhere in a program. As with Fortran typed data statements, field declarations can contain initializers. The name `%FILL` can be used in place of a field name to create space in the structure for padding. A `%FILL` space cannot be initialized.

A substructure must be declared by a RECORD statement that creates an instance of a previously declared structure, not by a nested STRUCTURE statement. The previous example shows the proper declaration of a substructure within the PERSON structure.

A union declaration defines a data area that can be shared during program execution and is bounded by UNION and END UNION statements. A union declaration must contain at least two map declarations (as indicated below) or an error condition occurs. A union declaration has the following form:

```
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 form:

```
MAP
      field declaration
        .
        .
        .
END MAP
```


VAX floating point data

CONVEX Fortran can read VAX floating point data in any format using the vax_d (vax-d) or vax_g (vax-g) data format names. The correct data format name is specified in the program's OPEN statement or in a shell variable. Chapter 9, "Input/output statements," explains in detail how data format names can be specified.

Table 37 lists various VAX floating point formats, their respective CONVEX equivalents, and the correct CONVEX data format name for reading and writing the format.

Table 37 VAX floating point data format names

VAX floating point format	CONVEX equivalent	CONVEX data format name
F_floating	REAL*4	vax_g, vax-g, vax_d, or vax-d
D_floating	REAL*8†	vax_d or vax-d
G_floating	REAL*8†	vax_g or vax-g
H_floating	REAL*16††	vax_g or vax-g
F_floating	COMPLEX*8	vax_g, vax-g, vax_d, or vax-d
D_floating	COMPLEX*16	vax_d or vax-d
G_floating	COMPLEX*16	vax_g or vax-g

† All REAL*8 data in a particular datafile must be read/written in either vax_d or vax_g format. The formats can not be mixed.

††The REAL*16 data type is supported on SPP Series machines and in native floating point mode on C Series machines.

Supported VAX intrinsics

The following VAX intrinsics are available when the `-vfc` compiler option is specified on the `fc` command line:

ACOSD	AIMAX0	AIMIN0	AJMAX0	AJMIN0
AKMAX0	AKMIN0	ASIND	ATAN2D	ATAND
BITEST	BJTEST	BKTEST	CDABS	CDCOS
CDEXP	CDLOG	CDSIN	CDSQRT	COSD
DACOSD	DASIND	DATAN2D	DATAND	DBLEQ
DCMPLX	DCONJG	DCOSD	DFLOAT	DFLOTI
DFLOTJ	DFLOTK	DIMAG	DIMAX0	DIMIN0
DJMAX0	DJMIN0	DKMAX0	DKMIN0	DMAX0
DMIN0	DREAL	DSIND	DTAND	FLOATI
FLOATJ	FLOATK	IDMAX1	IDMIN1	IIABS
IIAND	IIBCLR	IIBITS	IIBSET	IIDIM
IIDINT	IIDMAX1	IIDMIN1	IIDNNT	IIEOR
IIFIX	IINT	IIOR	IIQINT	IIQNNT
IISHFT	IISHFTC	IISIGN	IMAX0	IMAX1
IMIN0	IMIN1	IMOD	ININT	INOT
INT1	INT2	INT4	INT8	IQINT
IQNINT	IZEXT	JIABS	JIAND	JIBCLR
JIBITS	JIBSET	JIDIM	JIDINT	JIDMAX1
JIDMIN1	JIDNNT	JIEOR	JIFIX	JINT
JIOR	JIQINT	JIQNNT	JISHFT	JISHFTC
JISIGN	JMAX0	JMAX1	JMIN0	JMIN1
JMOD	JNINT	JNOT	JZEXT	KIABS
KIAND	KIBCLR	KIBITS	KIBSET	KIDIM
KIDINT	KIDMAX1	KIDMIN1	KIDNNT	KIEOR
KIFIX	KINT	KIOR	KIQINT	KIQNNT
KISHFT	KISHFTC	KISIGN	KMAX0	KMAX1
KMIN0	KMIN1	KMOD	KNINT	KNOT
KZEXT	QABS	QACOS	QACOSD	QASIN
QASIND	QATAN	QATAN2	QATAN2D	QATAND
QCOS	QCOSD	QCOSH	QDIM	QEXP
QEXT	QEXTD	QFLOAT	QFLOTI	QFLOTJ
QFLOTK	QINT	QLOG	QLOG10	QMAX1
QMIN1	QMOD	QNINT	QSIGN	QSIN
QSIND	QSINH	QSQRT	QTAN	QTAND
QTANH	SIND	SNGLQ	TAND	ZEXT

HP Fortran compatibility

F

This appendix describes the compatibility between Hewlett-Packard Fortran (HP Fortran) and CONVEX Fortran. The features described in this appendix are provided to facilitate porting code written for the HP Fortran compiler.

The CONVEX Fortran compiler supports the following features of the HP Fortran compiler:

- stack-based storage of uninitialized local variables
- use of intrinsics in constant expressions
- "loose" packing of COMMON block data items
- external naming conventions for Fortran program blocks

Each of these features is described in more detail in the sections that follow. In addition, CONVEX Fortran permits some HP Fortran compiler directives by disregarding them.

Local variable storage

By default in CONVEX Fortran (and HP Fortran) uninitialized local variables in subroutines and functions are allocated storage on the runtime stack. You can instruct the compiler to allocate static storage for uninitialized local variables either by specifying the `-nore` compiler option, or by using the `STATIC` storage class.

The `-nore` option, which is the default on SPP Series machines, instructs the compiler to store all variables in static storage. The `STATIC` storage class provides a way to force the compiler to allocate static storage for a specified list of variables.

Another storage class, `AUTOMATIC`, forces the compiler to allocate stack-based storage to a specified list of variables.

Constant expressions

CONVEX Fortran supports the use of some intrinsic routines in constant definitions. For a list of the intrinsics permitted in constant expressions and examples of their use, refer to the "Constants" section of Chapter 2.

COMMON block packing

Two methods are available for storing COMMON blocks: "loose" packing, which aligns all COMMON block data items to natural boundaries, and "tight" packing, which permits COMMON data items to align to unnatural boundaries. CONVEX Fortran supports loose packing for improved performance in certain situations and for compatibility with HP Fortran code.

By default on SPP Series machines, and when the `-align spp` compiler option is specified on C Series machines, the CONVEX Fortran compiler provides loose COMMON block packing by padding data items to their natural boundaries where necessary. This can improve performance in cases where partial-word items appear in COMMON blocks.

On C Series machines, or when the `-align cseries` compiler option is specified on SPP Series machines, the compiler stores COMMON blocks using tight packing (the ANSI standard). Because this method does not pad COMMON block data items, partial-word items may cause other data items to align on unnatural boundaries.

To optimize memory accesses in programs compiled with the `-align cseries` option, data items appearing in COMMON blocks should be ordered from largest to smallest; quad-word (16-byte) items should appear first (quad-word items are available only on C Series machines), followed by double-word items, single-word items, and finally partial-word items.

External naming conventions

The CONVEX Fortran compiler generates external names for user-written subprograms and COMMON blocks; these external names differ from the symbolic names used to reference subprograms and COMMON blocks in Fortran source code.

If you code part of your program in a language other than Fortran, such as C or assembly language, you must reference the external names generated by Fortran or the program cannot link properly.

CONVEX Fortran uses the naming conventions shown in Table 38, where *name* represents the symbolic name.

Table 38 CONVEX Fortran external naming conventions

Fortran program block	C Series external name	SPP Series external name
Main program	`_MAIN__`	`main__`
Blank COMMON	`___blnk_`	`_BLNK`
Named COMMON	*__name_*	*name*
Subprogram	*_name_*	*name*

As Table 38 shows, different external names are generated for CONVEX C Series and SPP Series computers.

To correctly reference the external names generated by Fortran, you must prepend and append the appropriate number of underscores.

Fortran User's Guide Chapter 4 describes Fortran's naming conventions in more detail. This chapter also contains several interlanguage programming examples that illustrate how to reference external names from both Fortran and C programs.

The `-noU77` naming option (SPP Series only)

By default, the compiler appends an underscore character (_) to names of routines in the `libU77` Fortran library (or any variety of `libU77`, such as `libU77p8`); this provides names that are distinct from the routine names generated by other languages, such as C. When the `-noU77` compiler option is specified, the compiler suppresses trailing underscores normally added to `libU77` names. The `-noU77` compiler option does not affect the way in which user-written subprograms are named.

Note

You must compile all sections of your program using the same naming convention option (either `-noU77` or the default) to ensure correct referencing of program blocks throughout the program.

The `-ppu` naming option (SPP Series only)

Available only on SPP Series machines, the `-ppu` compiler option forces the compiler to append an underscore character (_) to the end of external name definitions and references.

Note

You must compile all sections of your program using the same naming convention option (either `-ppu` or the default) to ensure correct referencing of program blocks throughout the program.

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. Table 39 lists the supported sequences.

Table 39 Supported Sun Fortran escape sequences

Character	Sequence
newline	`\n`
tab	`\t`
form feed	`\f`
null	`\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, which are implemented by CONVEX Fortran, represent a subset of Sun Fortran features. The Sun definition of `REAL` constants as 64-bit numbers is not supported.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

W

X

Z

ORDER NUMBER
DSW-037

DOCUMENT NUMBER
720-002230-011