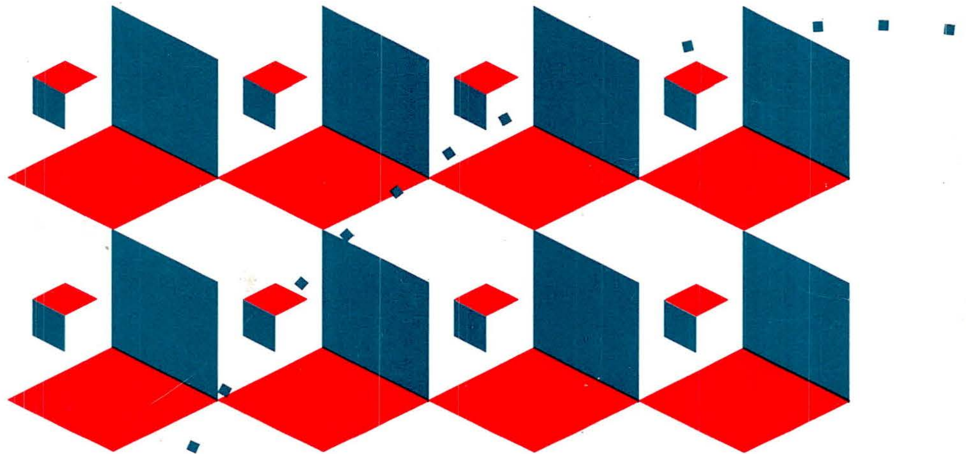
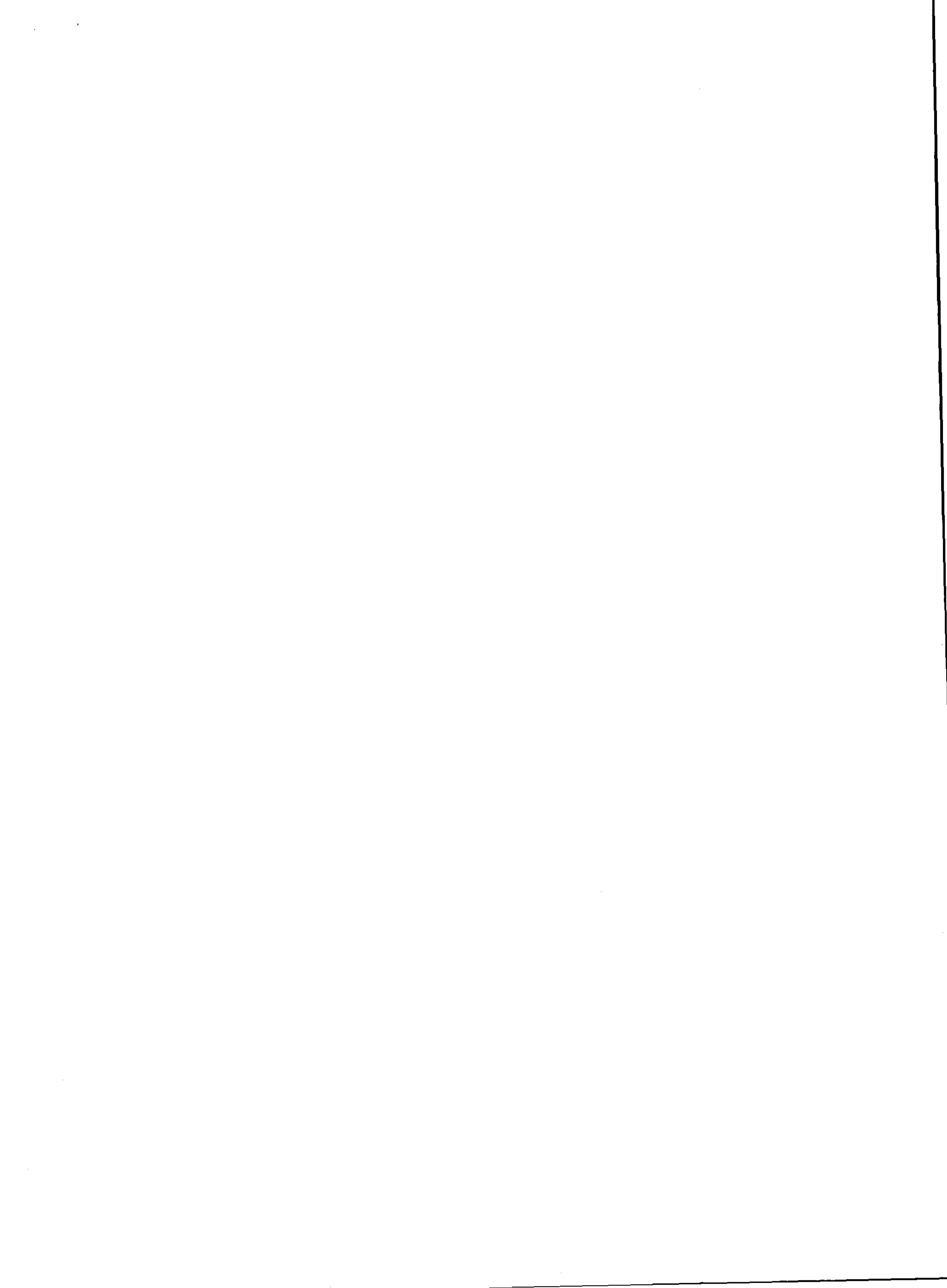


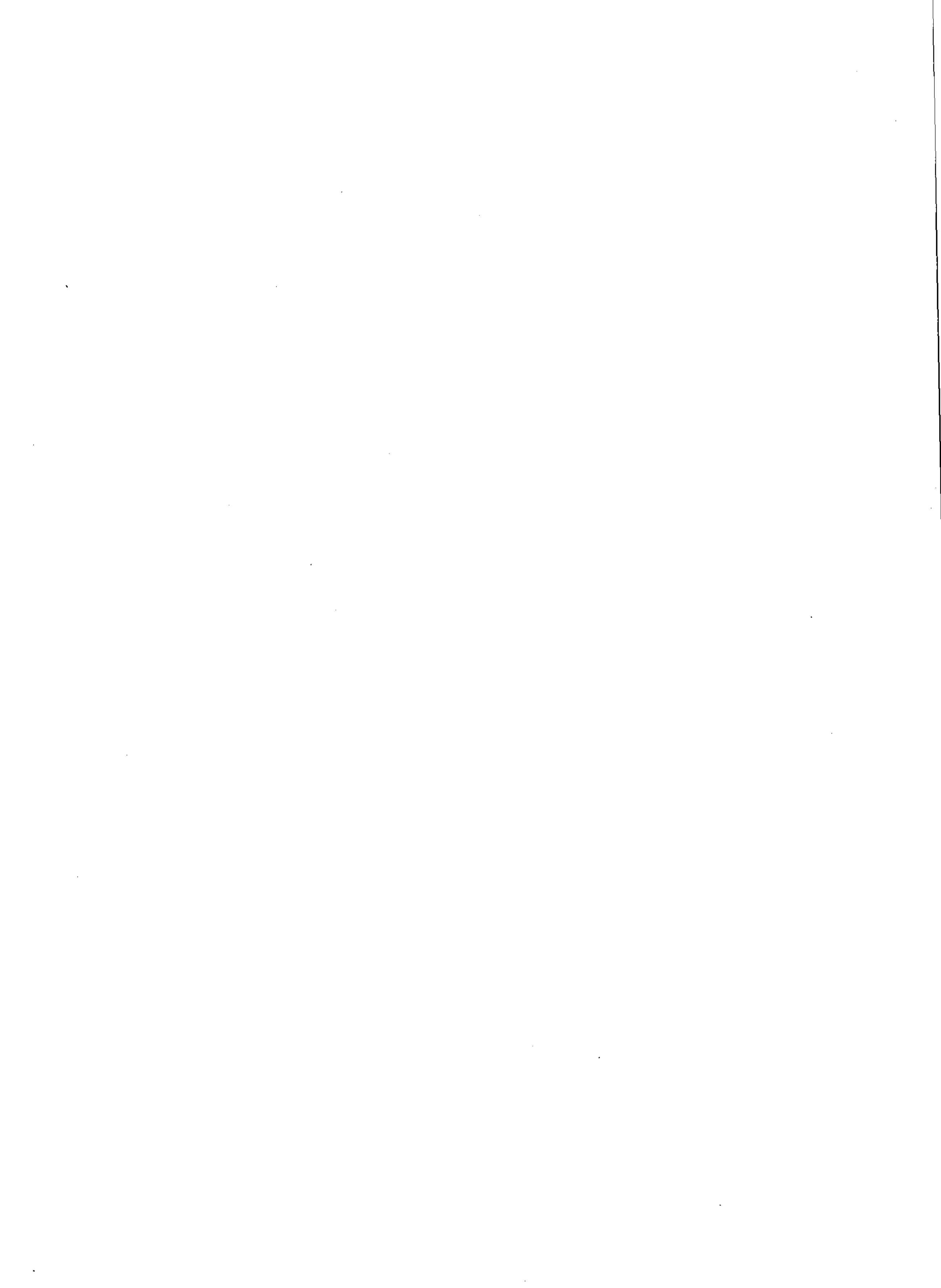
CONVEX

Fortran User's Guide

Eleventh Edition



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



Fortran User's Guide

Order No. DSW-038

Eleventh Edition
October 1994

CONVEX Press
Richardson, Texas
United States of America

Fortran
User's Guide

Order No. DSW-038

Copyright © 1994 CONVEX Computer Corporation
All rights reserved.

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

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

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

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
COVUE is a trademark of CONVEX Computer Corporation. COVUE products consist of COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.
UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc.

This entire book is recyclable.

Printed in the United States of America

Revision information for

Fortran User's Guide

Edition	Document No.	Description
Eleventh	720-000030-215	Released with CONVEX Fortran software V9.1, October 1994.
X11.0.0.2	720-000030-214	Released with CONVEX Fortran software V9.0.0.2, June 1994.
X11.0.0.1	720-000030-213	Released with CONVEX Fortran software V9.0.0.1, May 1994.
Tenth	720-000030-221	Released with CONVEX Fortran software V8.0, November 1992. Split *CONVEX Fortran Guide* into its component books. Extensively reorganized and expanded appendixes. Imported non-reference information from *CONVEX FORTRAN Language Reference Manual*. Added fcxref enhancements to Chapter 4. Added descriptions of CXmetrics and Application Compiler to Chapter 4. Added new compiler options to Chapter 1. Revised optimization report information in Chapter 1. Added new compiler directives to Appendix A. Added compiler error messages to Appendix C. Included CONVEX Fortran man pages as Appendix I.
Ninth Rev. 2	720-000030-208	Released with CONVEX Fortran V6.1, October 1990.
Ninth Rev. 1	720-000030-206	Released with CONVEX Fortran V6.0, March 1990.
Ninth	720-000030-204	Released with CONVEX Fortran V5.1, May 1989.
Eighth	720-000030-203	Released with CONVEX Fortran V5.0, November 1988.
Seventh Rev. 1	720-000030-202	Released with CONVEX Fortran V4.1, May 1988.
Seventh	720-000030-201	Released with CONVEX Fortran V4.0, November 1987.
Sixth	720-000030-200	Released with CONVEX Fortran V3.0, May 1987.
Fifth	720-000130-000	Released with CONVEX Fortran V2.2, September 1986.

Edition	Document No.	Description
Fourth	720-000130-000	Released with CONVEX Fortran V2.1, April 1986.
Third	720-000130-000	Released with CONVEX Fortran V1.7, October, 1985.
Second	720-000099-000	Released with CONVEX Fortran V1.6, August, 1985.
First	720-000130-101	Released with CONVEX Fortran V1.0, February 1985.

Contents

Figures xiii

Tables xv

How to use this guide xvii

Purpose and audience xvii
Organization xvii
Scope xviii
Notational conventions xviii
Associated documents xix
  C Series and SPP Series publications xix
  C Series publications xx
  SPP Series publications xxi
  Other documents xxi
  Online man pages xxi
Technical assistance xxii
The `contact` utility xxii

1 Compiling programs 1

Overview 1
File-naming conventions 3
Compiling programs 4
  Using compiler options 4
    `OPTIONS` statement 5
    `FCOPTIONS` environment variable 6
Compiler options 7
  Language-compatibility options 8
  Optimization options 9
  Code-generation options 15
  Debugging and profiling options 19
  Message and listing options 22
  Preprocessor options 25
  Miscellaneous options 26
Loading programs 29
  Using the `fc` command 29
Executing programs 30

Messages 30
Compiler messages 30
Runtime error messages 31
Optimization report 31
Loop table 32
Privatization table 32
Array table 32
Program interfaces 32
External naming conventions 33

2 Program development tools 35

Cross-reference generator 35
Cross-referencer options 36
Cross-referencing with -xr 38
Cross-referencing with -xra 39
Cross-reference report 40
Cover page 40
Routine reports 41
Module interface reports 47
Caller/callee routine cross-reference 51
Composite COMMON block reports 52
Include file reference table 55
Table of contents 55
Files and required utilities 56
Assembly- language debugger 56
Visual debugger (CXdb) 57
Performance analyzer (CXpa) 58
Profilers 59
CXtrace (SPP Series only) 60
Consultant III and CXtools 61
Application Compiler 61
Postmortem dump 62
(C Series only) 62
error utility 63

3 Input/output operations 65

Overview of I/O 65
C Series and SPP Series I/O differences 66
Supported I/O statements 66
Supported I/O methods 67
Records, files, and units 68
Records 68
Formatted records 69
Unformatted records 69
ENDFILE record 69
Files 70

Internal files 70
Units 70
Implicit unit numbers 71

4 Calling conventions.....75

Fortran subprogram calling convention 75
Fortran argument packets 75
Argument-passing mechanisms 77
Argument packet built-in functions 78
%REF function 78
%VAL function 79
Function return values 79
%LOC function 80
Non-Fortran-to- Fortran calling sequence 80
External naming conventions 82
C Series external naming 83
SPP Series external naming 83
The -noU77 naming option (SPP Series only) 83
The -ppu naming option (SPP Series only) 84
Data representations 84
Return values 85
Argument packets (C Series only) 86
Examples (C Series only) 87
Equivalent Fortran and non-Fortran code (C Series only) 87
Example 1 — simple procedure 87
Example 2 — two character arguments 88
Example 3 — character function 89
Example 4 — COMPLEX function 90
Example 5 — argument passing 90
Example 6 — array arguments 91
Interlanguage programming examples (C Series only) .. 92
Example 1 — passing INTEGER and REAL data 92
Example 2 — passing COMPLEX data 93
Example 3 — passing CHARACTER data 94
Example 4 — accessing COMMON blocks 95
Exemple 5 — REAL and INTEGER functions 96
Example 6 — string functions 97

5 System utilities.....99

How to call utility routines 99
Operating system utilities 100
The system utility 103
VAX-11 Fortran system utilities 104
date 104
idate 104

errsns 104
exit 104
secnds 105
time 105
ran 105
mvbits 105

6 Runtime errors and exceptions 107

I/O error processing 107
ERR and END specifiers 108
IOSTAT specifier 108
Signals and exceptions 109
Signals 109
Exceptions 112
Error-processing routines 113
setjmp and longjmp 113
errtrap (C Series only) 114
Hardware differences 115
signal 118
traceback 118
The perror, gerror, and ierrno routines 119
Examples of signal handling 120

A Compiler directives 123

Using compiler directives 123
Descriptions of compiler directives 125
BARRIER (SPP Series only) 125
BEGIN_TASKS, NEXT_TASK, END_TASKS 125
BLOCK_LOOP (SPP Series only) 127
BLOCK_SHARED (SPP Series only) 127
CRITICAL_SECTION, END_CRITICAL_SECTION (SPP Series only) 128
FAR_SHARED (SPP Series only) 128
FAR_SHARED_POINTER (SPP Series only) 129
FORCE_PARALLEL (C Series only) 129
FORCE_PARALLEL_EXT (C Series only) 130
FORCE_VECTOR (C Series only) 131
GATE (SPP Series only) 131
LOOP_PARALLEL (SPP Series only) 132
LOOP_PRIVATE 134
MAX_TRIPS (C Series only) 135
NEAR_SHARED (SPP Series only) 135
NEAR_SHARED_POINTER (SPP Series only) 136
NO_BLOCK_LOOP (SPP Series only) 136
NODE_PRIVATE (SPP Series only) 136
NODE_PRIVATE_POINTER (SPP Series only) 137

NO_LOOP_DEPENDENCE (SPP Series only) 137
NO_PARALLEL 138
NO_PEEL 138
NO_PROMOTE_TEST 138
NO_RECURRENCE (C Series only) 138
NO_SIDE_EFFECTS 139
NO_UNROLL_AND_JAM 140
NO_VECTOR (C Series only) 140
ORDERED_SECTION, END_ORDERED_SECTION (SPP Series only) 141
PEEL 141
PEEL_ALL 141
PREFER_PARALLEL 141
PREFER_PARALLEL_EXT (C Series only) 144
PREFER_VECTOR (C Series only) 145
PROMOTE_TEST 145
PROMOTE_TEST_ALL 145
PSTRIP (C Series only) 145
ROW_WISE 146
SAVE_LAST (SPP Series only) 147
SCALAR 148
SELECT (C Series only) 148
SYNCH_PARALLEL (C Series only) 149
TASK_PRIVATE 150
THREAD_PRIVATE (SPP Series only) 151
THREAD_PRIVATE_POINTER (SPP Series only) 151
UNROLL 152
UNROLL_AND_JAM 152
VSTRIP (C Series only) 153

B Fortran data representations 155
Logical representation 155
Integer representation 156
Real data representation 156
Native floating-point (C Series only) 157
IEEE floating-point 159
Complex representation 160
Character representation 160
Hollerith representation 161

C Compiler messages 163
Redirecting compiler messages 163
Compiler error messages 164

D Runtime error messages 209

System-detected errors 209
Runtime I/O errors (C Series) 209
Runtime I/O errors (SPP Series) 214

E Runtime libraries 219

Intrinsic library and math library 222
- Calling conventions 222
- Function-naming convention (C Series only) 222
- Intrinsic runtime routines 224
- Exponentiation programmed operators 243
- Complex programmed operators 245
- `REAL*16` programmed operators 246
- Vector mask programmed operators (C Series) 247
- String-manipulation programmed operators 247
- Runtime routine data items (C Series) 248

Fortran I/O library 248
- I/O operation 249
- I/O runtime routine naming convention (C Series) 249
- I/O list initialization 250
- I/O list element transmission 250
- I/O list termination 251
- Auxiliary I/O operations 252

F IEEE compatibility 253

G System limits 255

H ASCII character set 257

I Preprocessor 261

Preprocessor statements 261
- `#define` statement 261
- `#undef` statement 262
- `#include` statement 262
- `#if` statement 262

Preprocessor options 263
Preprocessor messages 263
User-defined preprocessors 263
- Sample preprocessor 264

Index 265

Figures

Figure 1	Default `fc` compilation process	2
Figure 2	Cross-referencer report cover page	41
Figure 3	Cross-referencer routine report	44
Figure 4	Cross-referencer routine report structure summary	46
Figure 5	Cross-referencer routine COMMON block summary	47
Figure 6	Module interface section example	49
Figure 7	Caller/callee routine cross reference example	52
Figure 8	Cross-referencer COMMON block summary example	53
Figure 9	Include file reference table example	55
Figure 10	Cross-referencer table of contents example	56
Figure 11	Argument packet: example 1	76
Figure 12	Argument packet: example 2 (C Series only)	77
Figure 13	Calling a Fortran subroutine (C Series only)	81
Figure 14	C1 Series intrinsic errors	117
Figure 15	C2, C3, and C4 Series intrinsic errors	117
Figure 16	`traceback` resulting from an exception	118
Figure 17	`traceback` generated by a user-called subroutine	119
Figure 18	Signal handler for interrupts	120
Figure 19	Signal handler for arithmetic exceptions	121
Figure 20	`LOGICAL` data type representation	155
Figure 21	`INTEGER` data type representation	156
Figure 22	`REAL` data type representation	157
Figure 23	`COMPLEX` data type representation	160
Figure 24	`CHARACTER` data type representation	161

Tables

Table 1 File name extensions....4
Table 2 OPTIONS statement compiler options....5
Table 3 C Series-specific compiler options (C Series only)....7
Table 4 SPP Series-specific compiler options (SPP Series only)....7
Table 5 Cross-referencer (fcxref) options....23
Table 6 CONVEX Fortran external naming conventions....33
Table 7 Cross-referencer options....37
Table 8 fcxref context operators....43
Table 9 -xrm *n* options....48
Table 10 Compiler options for profiling....60
Table 11 Postmortem dump contents....62
Table 12 Input/output statements....66
Table 13 Input/output methods....67
Table 14 C Series implicit unit numbers (C Series only)....72
Table 15 SPP Series implicit unit numbers (SPP Series only)....72
Table 16 Implicit units numbers by Fortran statement....73
Table 17 Built-in functions and defaults for argument lists....78
Table 18 C Series function return values (C Series only)....79
Table 19 SPP Series function return values (SPP Series only)....80
Table 20 CONVEX Fortran external naming conventions....82
Table 21 CONVEX C external naming conventions....83
Table 22 Fortran and C declarations....84
Table 23 Complex function: C equivalent (C Series only)....85
Table 24 Character functions: C equivalent (C Series only)....86
Table 25 Character arguments: C equivalent (C Series only)....87
Table 26 Calling sequences for ConvexOS and SPP-UX utilities....100
Table 27 C Series signal names and numbers (C Series only)....110
Table 28 SPP Series signal names and numbers (SPP Series only)....111

Table 29 C Series mapping of exceptions to signals and codes (C Series only) 112
Table 30 SPP Series mapping of exceptions to signals and codes (SPP Series only) 113
Table 31 errtrap argument flags........ 115
Table 32 Intrinsic instructions—C200, C3200, C3400, C3800, and C4600 Series 116
Table 33 Compiler directives available on C Series and SPP Series machines124
Table 34 Maximum parallel strip-mine lengths at -O3 146
Table 35 Maximum vector strip-mine lengths at -O3........ 154
Table 36 Four processor system strip lengths 154
Table 37 Operands defined by CONVEX native floating-point........ 158
Table 38 Operands defined by CONVEX IEEE floating point 159
Table 39 Fortran runtime libraries (C Series) 220
Table 40 Fortran runtime libraries (SPP Series) 221
Table 41 Argument/result codes 223
Table 42 Intrinsic functions........ 224
Table 43 Exponentiation routines 244
Table 44 Complex programmed operators........ 245
Table 45 REAL*16 programmed operators 246
Table 46 ASCII character set 257
Table 47 System limits for C Series and SPP Series programs........ 255

How to use this guide

Purpose and audience

This guide describes how to use the CONVEX Fortran compiler on both CONVEX C Series and CONVEX SPP Series machines. Some information is specific to one hardware platform and is marked to indicate this within the text.

Subjects discussed in this guide include compiling, loading, and executing programs. Other pertinent information includes input/output operations, error processing, program optimization, utility libraries, and debugging.

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 guide is organized as follows:

- Chapter 1 has an overview of the CONVEX Fortran compiler and describes how to compile, load, and execute a program. The chapter also includes an overview of the optimizations available in the compiler.
- Chapter 2 presents an overview of the tools available for debugging and analyzing Fortran programs.
- Chapter 3 presents an overview of the key input/output concepts and features of CONVEX Fortran.
- Chapter 4 describes the CONVEX Fortran calling conventions and describes how to call routines written in languages other than Fortran.

- Chapter 5 describes the use of system utilities for the C Series and SPP Series operating systems, ConvexOS and SPP-UX.
- Chapter 6 discusses runtime error processing and describes how the runtime library processes errors, what the defaults are, and how to override the defaults.
- Appendix A lists compiler directives supported by CONVEX Fortran and describes their use.
- Appendix B describes the data types supported by CONVEX Fortran and shows their internal representations.
- Appendix C lists error and advisory messages that can occur during compilation.
- Appendix D lists error and advisory messages that can occur at runtime.
- Appendix E lists and describes the runtime library and routines.
- Appendix F describes CONVEX Fortran compatibility with the IEEE 754 standard.
- Appendix G lists system limits.
- Appendix H lists the ASCII character set.
- Appendix I describes the CONVEX Fortran preprocessor.

Scope

This guide 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.

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 applying to all 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.

- *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.*

- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251) (optional product) describes how to use the interactive profiler.
- *Fortran Language Reference* (DSW-037) describes the Fortran language and CONVEX extensions to the language.

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.

- *CONVEX adb Debugger User's Guide* (DSW-009), a tutorial and reference manual, describes the functions and operations of the CONVEX `adb` debugger.
- *CONVEX Application Compiler User's Guide* (DSW-401) (optional product) describes how to use the CONVEX Application Compiler to optimize programs.
- *CONVEX Compiler Utilities User's Guide* (DSW-096) describes the CONVEX loader and the CONVEX assembler.
- *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 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 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.
- *CXmetrics User's Guide* (DSW-475) (optional product) describes software metrics data and explains how to use CONVEX CXmetrics to report on this data.
- *Fortran Optimization Guide* (DSW-034) describes the types of optimization available in CONVEX Fortran and shows you how to use optimization directives and options.

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 Architecture* (DHW-014) describes the architecture of Exemplar (SPP Series) computers.
- *Exemplar Programming Guide* (DSW-067) describes efficient programming techniques for the Exemplar family (SPP Series) of computers.
- *HP-UX System Administration Guide* (Hewlett-Packard order number B2355-90004) discusses material pertinent to SPP-UX.
- *SPP-UX System Administration Guide* (DSW-853) provides information on administering SPP-UX.

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 man 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.

Compiling programs

1

This chapter presents an overview of the CONVEX Fortran compiler and explains how to compile, load, and execute programs written in CONVEX Fortran.

CONVEX Fortran is a high-level programming language available for CONVEX C Series machines and CONVEX SPP Series machines.

CONVEX Fortran includes standard Fortran as defined by the American National Standard FORTRAN 77 (ANSI X3.9-1978). It also includes selected Fortran 90 extensions; VAX-11 features; certain features of Cray, Sun, and Hewlett-Packard Fortran; and unique CONVEX extensions. The *Fortran Language Reference* contains a complete description of the CONVEX Fortran language.

CONVEX Fortran is available for CONVEX C Series computers and CONVEX SPP Series computers (including Exemplar systems). Throughout this user's guide, some information is specific to either C Series or SPP Series machines and is marked to indicate this in the text.

Overview

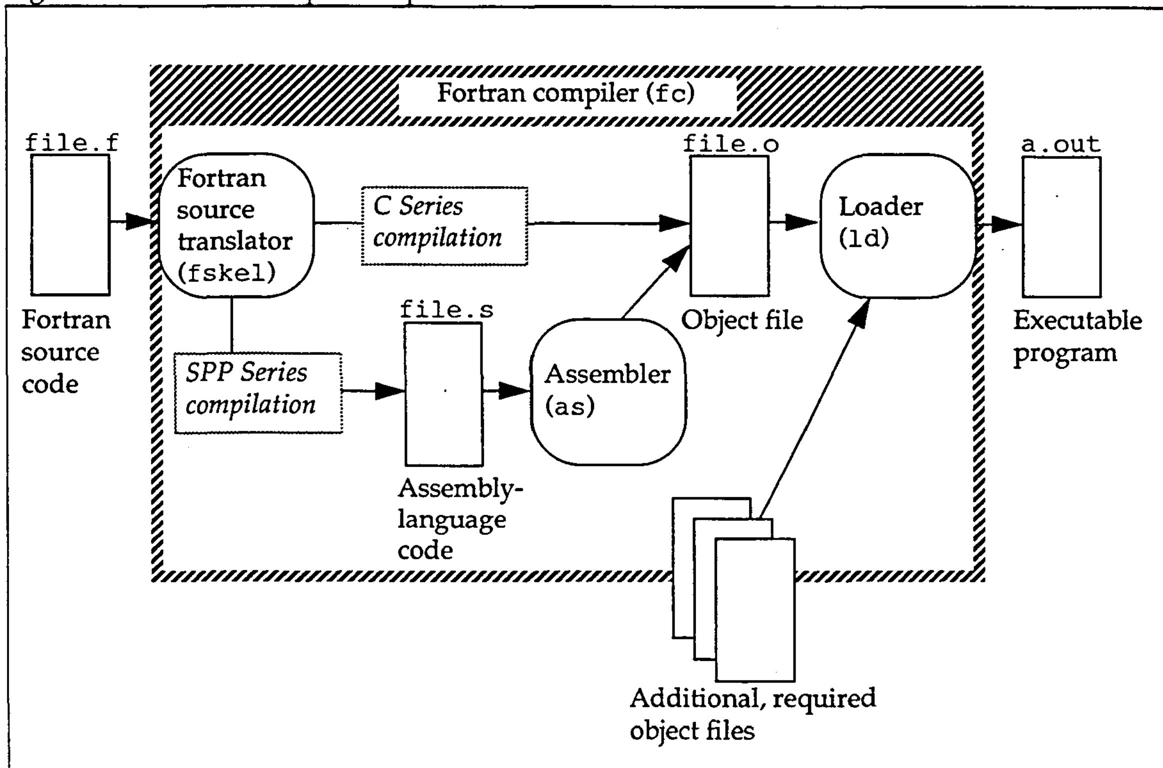
The CONVEX Fortran compiler translates a source file containing one or more Fortran program units into an object module. The object module then can be linked with library routines or other object modules for execution on a CONVEX computer. On C Series computers, previously compiled programs written in CONVEX assembly language, C, C++, or Ada can interface with CONVEX Fortran object code to produce an executable program. On SPP Series machines, previously compiled programs written in PA-RISC assembly, CONVEX C, C++, or any PA-RISC object file can interface with CONVEX Fortran object code to produce an executable program.

The CONVEX Fortran compiler automatically generates code that takes full advantage of the architecture of the CONVEX

supercomputer for which it is compiled. CONVEX Fortran is available for both the C Series and SPP Series architectures. In addition, use of optimization options causes the compiler to optimize, vectorize (only on CONVEX C Series machines), and parallelize source code to maximize execution efficiency.

To provide a fast, complete compilation, the compiler automatically assembles and loads files as needed to generate an executable version of your Fortran source code. The Fortran compilation process is normally driven by the `fc` program, which manages the compilation as shown in Figure 1. The `fc` driver also passes the proper options to the various executables invoked during the compilation process and performs other tasks. The default CONVEX Fortran compilation process on both C Series and SPP Series machines is illustrated in Figure 1.

Figure 1 Default `fc` compilation process



The compiler provides an option, `-S`, to produce only assembly-language code, which can be inspected, modified, or assembled directly using the assembler (`as`). Another compiler option, `-c`, generates an object file (also known as an object module), which the loader (`ld`) can combine with other object files to produce an executable program.

By default, the CONVEX Fortran compiler provides compatibility with certain features of HP Fortran.

The compiler also has options for requesting compatibility with Sun Fortran, Cray Fortran, and VAX Fortran, as well as options that provide additional compatibility with HP Fortran.

On C Series machines, the CONVEX Fortran compiler provides options for selecting either IEEE-standard or native representation of floating point values.

On CONVEX SPP Series machines, floating-point numbers comply with the IEEE standard (IEEE/ANSI standard 754-1985).

On both C Series and SPP Series, the compiler assists in program checkout by interfacing with several debuggers and utility routines, including a source-level debugger and an assembly-level debugger.

On C Series computers, CONVEX Fortran programs can optionally be compiled under COVUEshell. COVUEshell is a CONVEX product for C Series machines that provides a VMS-like interface and supports much of the Digital Command Language (DCL). For further information, refer to the *CONVEX COVUEshell Reference Manual.*

Also on C Series machines, the compiler provides an option for requesting inline substitution of subroutines. Inline substitution involves replacing a function or subroutine call with the actual body of the procedure, thus eliminating the overhead of a subprogram call and allowing additional optimization.

File-naming conventions

The compiler discerns a file type by the extension added to the file name. A Fortran source file is identified by either the extension `.f` or the extension `.FOR`, as in `source.f` or `prog1.FOR`. A compiled object file has the same name as its corresponding source file except its extension is `.o`. Symbolic assembly files have names with a `.s` extension. Unless you specify otherwise on the compiler command line, the executable module produced by the loader is placed into a file named `a.out`. The CONVEX Fortran compilation process is briefly summarized in Figure 1 on page 2.

Table 1 summarizes the file-naming conventions used by the CONVEX Fortran compiler.

Table 1 File name extensions

Type of file	File name extension
Fortran source files	`.f` or `.FOR`
Object files	`.o`
Symbolic assembly-language files	`.s`
Inline intermediate files	`.fil`

Compiling programs

The `fc` command compiles CONVEX Fortran source and has the following form on both CONVEX C Series and CONVEX SPP Series machines:

```
fc [options] files
```

On the `fc` command line, *options* is one or more of the compiler options described in the "Compiler options" section of this chapter. Options also may be specified in an `OPTIONS` statement within a program unit or through the `FCOPTIONS` environment variable. All three methods of selecting compiler options are discussed in the "Using compiler options" section that follows.

The *files* parameter represents one or more Fortran source files to be compiled, object files to be loaded, or symbolic assembly-language files to be generated.

Loader options are specified by using the `-W` compiler option, which is described later in the "Compiler options" section of this chapter. Refer to the "Loading programs" section of this chapter for information on using loader options.

Using compiler options

You can specify compiler options in three ways: on the `fc` command line, through the `FCOPTIONS` environment variable, and in an `OPTIONS` statement within a Fortran program unit.

The following is the order in which `fc` evaluates compiler options:

1. `FCOPTIONS` environment variable
2. Command line options
3. `OPTIONS` statement

When fc compiles a program, first the options specified via the FCOPTIONS environment variable are considered along with any command line options. Any conflicts among options specified by these two methods are reported and resolved. For example, when optimization level -O2 is selected by FCOPTIONS and level -O3 is requested on the command line, the following message results:

```
Contradictory optimization level specifications given -- believed '-O3'.
```

This message indicates that, when evaluating the command line and FCOPTIONS, more than one optimization level was specified and level -O3, the command line specification, ultimately was selected.

OPTIONS statement

The OPTIONS statement, if specified, is evaluated after command line options and the FCOPTIONS environment variable. Only the compiler options listed in Table 2 may appear in an OPTIONS statement.

Note

No warning message is provided when options in an OPTIONS statement conflict with those requested in FCOPTIONS or the command line; instead, when conflicts occur, the OPTIONS statement's options supersede those previously requested.

Compiler options specified in an OPTIONS statement have highest precedence. For example, when -O2 is specified through the FCOPTIONS variable, -O3 is specified on the fc command line, and -O1 is specified through an OPTIONS statement, the code is compiled at optimization level -O1.

Table 2 lists all compiler options that may appear in OPTIONS statements.

Refer to the "Compiler options" section of this chapter for more information about the options in Table 2, including restrictions associated with their use. Some of these options are available on only C Series or only SPP Series machines.

Table 2 OPTIONS statement compiler options

-al	-blockloop *n*	-cache *n*
-ds	-ep *n*	-F66
-i*n*	-na	-no

Table 2 (continued) `OPTIONS` **statement compiler options**

-noblock	-nopeel	-noptst
-nsr	-nuj	-nur
-nw	-O0	-O1
-O2	-O3	-peel
-peelall	-ptst	-ptstall
-r*n*	-re	-rl
-sa	-sr	-uj
-ujn *n*	-uo	-ur
-urn *n*		

FCOPTIONS environment variable

The environment variable FCOPTIONS can be used to create default fc options. These options will be automatically supplied each time fc is invoked. You can set FCOPTIONS under csh using the following form:

```
setenv FCOPTIONS "optionlist"
```

where *optionlist* is a list of one or more compiler options, separated by spaces, that you want fc to use every time it is invoked. Under csh, FCOPTIONS can be initialized in your .cshrc file if you always use the same options with fc.

If you use a shell other than csh, consult the shell's man page for information on setting environment variables under it.

The following csh example sets the FCOPTIONS environment variable to specify the -nw and -O3 compiler options.

```
setenv FCOPTIONS "-nw -O3"
```

All programs compiled with fc while this setting is active are compiled at optimization level -O3 and all warning diagnostic messages are suppressed.

You can check the current value of FCOPTIONS under csh by using the printenv command. This command prints the values of variables in the current shell environment. The following command under csh prints the value of FCOPTIONS if it is

defined, and otherwise prints nothing (thus indicating that FCOPTIONS is undefined and therefore does not affect Fortran compilations):

```
printenv FCOPTIONS
```

Compiler options

The sections that follow list the options available for use with the CONVEX Fortran compiler. You can specify these options to choose the way a program will be compiled. The previous section, "Compiling programs," describes how to specify the options that will be used when compiling a program.

Table 3 lists compiler options that are available only on CONVEX C Series machines.

Table 3 C Series-specific compiler options (C Series only)

-ansic	-ds	-ep *n*
-except default	-except precise	-fi
-fn	-il	-is *directory*
-metrics	-mi *n*	-nopm
-pb	-pcc	-sa
-tl *n*	-xrl	-xro

Table 4 lists compiler options that are available only on CONVEX SPP Series machines.

Table 4 SPP Series-specific compiler options (SPP Series only)

-blockloop *n*	-br	-cache *n*
-mo	-nbr	-nga
-ngs	-nmo	-noU77
-noblock	-ppu	

Some compiler options, such as -tm, have features that are available only on either C Series or SPP Series machines. These restrictions are noted in the descriptions of these options in the following sections.

Language-compatibility options

The options listed in this section provide support for various non-native Fortran compiler features.

`-ansi77`

Causes the compiler to perform strict checking for violations of the ANSI FORTRAN 77 standard. Code compiled with this option should not contain CONVEX extensions, Fortran 90 code, or any other extensions.

`-ansi90`

Causes the compiler to perform strict checking for violations of the ANSI Fortran 90 standard. For information about CONVEX Fortran compatibility with the Fortran 90 standard, refer to *Fortran Language Reference* Appendix C, "Fortran 90 Compatibility."

`-cfc`

Causes the compiler to accept and use certain syntax, defaults, and features of the Cray Fortran language. Store variables and arrays using Cray Fortran's data type lengths. This option implies `-pd8` and cannot be used with the `-i` or `-r` options. For a list of supported Cray Fortran features, consult *Fortran Language Reference* Appendix D, "Cray Fortran compatibility."

`-cfcwpa`

Causes the compiler to compute word addresses when performing Cray pointer arithmetic on explicitly declared Cray pointers. Byte addresses are still computed on pointers not explicitly declared using the `POINTER` statement. Attempting to perform an operation other than addition or subtraction on a declared Cray pointer under this option will cause the compiler to issue a warning message. `-cfcwpa` must be used with the `-cfc` option. Refer to Appendix D of the *Fortran Language Reference* for more information.

`-dfc`

Use `-dfc` only when you use `-vfc` and the Binary Data File Format Conversion feature discussed in Chapter 9 of the *Fortran Language Reference*. This option causes all references to VAX records in an I/O statement to be decomposed, and is useful only when doing I/O on VAX records.

`-F66`

Causes the compiler to accept and use certain syntax, defaults, and features of FORTRAN 66. Before using this option, read Appendix B of the *Fortran Language Reference.*

`-nof90`

Disables the Fortran 90 language compatibility normally provided by CONVEX Fortran. See *Fortran Language Reference* Appendix C for a summary of Fortran 90 features that CONVEX Fortran makes available by default.

C Series only

`-sa`

On C Series machines, prevents Fortran from generating precompiled argument packets in the text segment. All arguments are placed on the stack. This option should only be used when a Fortran program calls user-supplied C programs. Using it with an application coded only in Fortran slows down the application.

`-sfc`

Causes the compiler to accept and use certain syntax, defaults, and features of the Sun Fortran language. Before using this option, read Appendix G of the *Fortran Language Reference.*

`-vfc`

Causes the compiler to accept and use additional defaults of the VAX Fortran language. Before using this option, read Appendix E of the *Fortran Language Reference.*

Optimization options

The options listed in this section enable various optimizations designed to make code execute more efficiently.

SPP Series only

`-blockloop` *n*

Instructs the compiler to block *m*-1 loops in an *m* loop nest. The compiler uses a block factor of *n* and automatically selects which loops to block. For detailed information about loop blocking consult the *Exemplar Programming Guide.* Loop blocking is provided at optimization levels `-O2` and `-O3` unless the `-noblock` option is specified.

SPP Series only

`-br`

Enables base register optimizations. This option is the default at optimization levels `-O1` and higher. To disable base register optimizations use the `-nbr` compiler option.

SPP Series only

`-cache n`

Instructs the Fortran compiler to assume a single processor direct-mapped cache size of *n* kilobytes. The compiler uses this information when determining a loop's blocking factor. Loop blocking is provided at optimization levels `-O2` and `-O3` unless the `-noblock` option is specified.

C Series only

`-ds`

Causes the compiler to automatically select loops to replicate and to compile scalar, vector, and possibly parallel and parallel/vector versions of such loops. The compiler then dynamically selects the version of each loop to be executed. This option is available only on C Series machines at optimization levels `-O2` or `-O3`.

C Series only

`-ep` *n*

Specifies the expected number of processors (*n*) on which the program is going to run. The `-O3` option must be present on the command line for `-ep` to be effective. The value of *n* should be an integer from 1 to 8.

The compiler parallelizes a loop whenever doing so appears to decrease the turnaround time, assuming the given number of processors. Use this option with caution because it may lead to inefficient use of processors.

C Series only

`-il`

Instructs the compiler to prepare an intermediate language (`.fil`) file for a subprogram that is to be used for inline substitution. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization levels are ignored.

C Series only

`-is` *directory*

Instructs the compiler to attempt inline substitution of each subprogram for which there exists a `.fil` (intermediate-language file) file in the specified *directory*. This option must be repeated for each directory containing `.fil` files to be used for inline substitution.

SPP Series only

`-mo`

Enables the generation of multi-op instructions. The `-mo` option is the default on SPP Series machines. To prevent the generation of multi-op instructions specify the `-nmo` option.

SPP Series only

`-nbr`

Disables base register optimizations. By default, CONVEX Fortran provides base register optimizations at optimization levels `-O1` and higher.

SPP Series only

`-nga`

Disables global register allocation for arguments passed by reference. Global register allocation is enabled by default at optimization levels `-O1` and higher.

SPP Series only

`-ngs`

Disables global register allocation for shared memory variables which are visible to multiple threads. Global register allocation is enabled by default at optimization levels `-O1` and higher.

SPP Series only

`-nmo`

Disables the generation of multi-op instructions. Multi-op instructions are generated by default on SPP Series machines (and via the `-mo` option).

`-no`

Specifies that the compiler is to perform no optimization. This option is the default if the `-O`*n* option is not specified.

`-noautopar`

Parallelizes loops only if a parallelizing compiler directive (such as `PREFER_PARALLEL` or `BEGIN_TASKS`) precedes them to request parallelization. All other loops are treated as if the `NO_PARALLEL` directive were specified for them. This option affects optimization level `-O3`.

Under `-noautopar`, loop interchanges for parallelization occur only for loops that are affected by parallelization directives and are parallelized. This option does not interfere with loop blocking and other level `-O2` interchanges.

C Series only

`-noautovec`

Vectorize loops only if a vectorizing compiler directive (for example, `PREFER_VECTOR` or `FORCE_VECTOR`) precedes them to request vectorization. This option is effective at optimization levels `-O2` and `-O3`.

The compiler still interchanges loops that otherwise would have been fully (100%) vectorizable because of directive specifications or dependency deductions. Interchanges are not performed for partial vectorization. Any loop that is interchanged is vectorized only if a vectorization directive applies to the loop.

SPP Series only

`-noblock`

Prevents the compiler from blocking loops in the specified Fortran source files. Because loop blocking occurs at

optimization levels -O2 and -O3, the -noblock option is effective only at these levels.

-nopeel

Disallows loop boundary value peeling, which is enabled by default at optimization levels -O2 and -O3. Refer to the -peel and -peelall options described in this section. Refer also to the *Fortran Optimization Guide* for more C Series information. For SPP Series optimization information see the *Exemplar Programming Guide*.

C Series only

-nopm

Disables pattern matching used by the compiler to aid in vectorization. Normally the compiler uses pattern matching to vectorize loops not otherwise vectorizable. Vectorization is available only on CONVEX C Series machines. On SPP Series machines, the -nopm option is ignored.

-noptst

Disallows test promotion, which is enabled by default at optimization levels -O2 and -O3. Refer to -ptst and -ptstall below. Refer also to the *Fortran Optimization Guide* for more C Series information. For SPP Series optimization information see the *Exemplar Programming Guide*.

-nsr

Disables scalar replacement. Scalar replacement is enabled by default and affects optimization levels -O2 and -O3.

-nuj

Disables the loop unroll and jam transformation. On SPP Series machines, unroll and jam is enabled by default. When enabled, this transformation affects optimization levels -O2 and higher.

-nur

Disables loop unrolling On SPP Series machines, loop unrolling is enabled by default. Unrolling affects optimization levels -O2 and -O3.

-O*n*

Performs machine-independent optimizations at the specified level. You can specify the following optimization levels:

Level	Description
-O0	Basic block machine-independent scalar optimization.
-O1	Program unit level scalar optimizations and global register allocation.
-O2	Global instruction scheduling, software pipelining, and data localization optimizations. On C Series machines, vectorization also is provided.
-O3	Parallel optimizations.

If this option is not specified, no machine-independent optimization is performed. The default optimization level is -no, which provides scalar optimizations at the machine instruction level.

-peel

Removes the first and/or last iterations of a loop when doing so removes 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 to .TRUE. or .FALSE. for the first and/or last iteration. By default, the compiler peels boundary values and expands code up to a predetermined conservative limit. With the -peel option, this limit is increased and code expansion may become significant. This option is available only at optimization level -O2 or -O3. Refer to the *Fortran Optimization Guide* for additional C Series information. For SPP Series optimization information see the *Exemplar Programming Guide.*

-peelall

Same as -peel, but allows code expansion without bound. For code containing large numbers of boundary value operations, this 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. This option is available only at optimization level -O2 or -O3. Refer to the *Fortran Optimization Guide* for more C Series information. For

SPP Series optimization information see the *Exemplar Programming Guide.*

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticeable increase in compile time. This option is available only at optimization level `-O2` or `-O3`. Refer to the *Fortran Optimization Guide* for more C Series information. For SPP Series optimization information see the *Exemplar Programming Guide.*

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, 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. This option is available only at optimization level `-O2` or `-O3`. Refer to the *Fortran Optimization Guide* for additional C Series information. For SPP Series optimization information see the *Exemplar Programming Guide.*

`-rl`

Causes the compiler to automatically select loops and replicate them by unrolling or dynamic selection. This option is available only at optimization level `-O2` or `-O3`.

`-sr`

Enables scalar replacement, which involves storing loop-invariant array references in registers throughout a loop, thus avoiding slower main memory accesses. This option is enabled by default. When enabled, this replacement may occur at levels `-O2` and `-O3`. Scalar replacement is disabled by the `-nsr` option.

`-uj`

Enables the loop unroll and jam transformation, whose goal is to exploit the use of registers and thus decrease the number of slower main memory accesses. The `-uj` option is enabled by default on SPP Series machines. Unroll and jam is effective at optimization levels `-O2` and `-O3`. This transformation is disabled via the `-nuj` option. To specify the loop unrolling factor for this transformation use the `-ujn` *n* option. For more SPP Series information consult the *Exemplar Programming Guide.*

`-ujn n`

Enables the loop unroll and jam transformation and specifies the desired loop unrolling factor (*n*, where *n* is the number of times to replicate the body of the loop). This option, when specified, is effective at optimization levels `-O2` and `-O3`.

`-uo`

Performs potentially unsafe optimizations, for example, moving the evaluation of common subexpressions or invariant code from within conditionally executed code. This moved code may be executed unconditionally.

`-ur`

Causes the compiler to automatically select and unroll (or partially unroll) loops. On SPP Series machines, loop unrolling occurs by default. Loop unrolling affects optimization levels `-O2` and `-O3`. To disable `-ur` specify the `-nur` option.

`-urn n`

Causes the compiler to automatically select and unroll loops, and use a loop unrolling factor of *n*, where *n* is the number of times to replicate the body of the loop. This option is available at optimization levels `-O2` and `-O3`.

Code-generation options

The options listed in this section control various aspects of code generation, including default variable and constant sizes.

`-c`

Suppresses the loading phase of the compilation. For example, output from the file `file.f` or `file.s` is written to `file.o`. See the "Overview" section of this chapter for an overview of the Fortran compilation process.

C Series only

`-except precise`

Generates code that ensures that any arithmetic exceptions caused by a subprogram before it returns will be received by the subprogram. Without `-except precise`, there is a small possibility that the location of an arithmetic exception might be reported incorrectly or lost.

The code generated under `-except precise` is specific to the target architecture for which you are compiling and is only guaranteed to work for that architecture. The target

architecture is determined by the `-tm` option, or, in absence of `-tm`, defaults to the machine on which you are compiling.

Use `-except precise` only when absolutely necessary as it causes additional instructions to be inserted before every return, and this will degrade performance.

C Series only

`-except default`

Cancels the effects of `-except precise`. Use this option to override an `-except precise` supplied in the `FCOPTIONS` environment variable.

C Series only

`-fi`

Specifies that real constants are to be translated into IEEE format and processed in IEEE mode. If you specify this option, your machine must be equipped with IEEE support hardware, or an error message occurs and compilation terminates. If you do not specify a floating-point format, your site default is used. This option cannot be used with `REAL*16` data.

CONVEX hardware and software only support the processing of data encoded in IEEE format and do not fully conform to the IEEE 754 specifications for arithmetic (see Appendix D, "IEEE compatibility").

C Series only

`-fn`

Specifies that real constants are to be translated into native format and processed in native mode. If you do not specify a floating-point format, your site default is used.

`-i`*n*

Specifies that `INTEGER` and `LOGICAL` variable declarations with unspecified lengths are to occupy *n* bytes of storage, where *n* can be 2, 4 (the default), or 8. Transforms intrinsic function references that return default integer or logical values to return integer values of the specified length. Storage association between integer and real data is not maintained as they are defined by ANSI FORTRAN 77.

Note

The `-i`*n* option is supported for compatibility with previous releases only and has been replaced by the `-p8` and `-pd8` options. It should not be used.

C Series only

`-mi` *n*

Specifies the expected memory interleave on the target machine. *n* is an integer representing the expected memory interleave, which you can obtain for your machine with the

getsysinfo command. When this option does not appear, the interleave of the machine the compiler is running on is used.

-nore

Causes the compiler to generate code that passes arguments to subroutines using argument packets instead of the stack. This allows local subroutine and function variables to maintain their values between calls. This option overrides the default on SPP Series machines, which is to generate reentrant code for parallel or recursive invocation of subroutines by passing arguments on the stack. The -nore option is the default on C Series computers.

-p8

Specifies that INTEGER, LOGICAL, and REAL variables and constants with unspecified lengths are to occupy 8 bytes of storage; DOUBLE PRECISION and COMPLEX values are to occupy 16 bytes of storage. Transforms intrinsic function references that take or return default INTEGER, LOGICAL, REAL, COMPLEX, DOUBLE-PRECISION values to take values of the specified length. Storage association and intrinsic function references maintain full compatibility with ANSI FORTRAN 77. DOUBLE PRECISION variables and constants are not allowed if IEEE format is requested.

You can override the default size of any variable with length override specifications in its declaration. You can also use intrinsic functions that are defined to take or return INTEGER, REAL, COMPLEX, or LOGICAL values of a specific length.

-pd8

Specifies that INTEGER, LOGICAL, REAL, and DOUBLE PRECISION variables and constants with unspecified lengths are to occupy 8 bytes of storage; COMPLEX values are to occupy 16 bytes. Transforms intrinsic function references that take or return default INTEGER, LOGICAL, REAL, COMPLEX, or DOUBLE PRECISION values to take values of the specified length. Intrinsic function references maintain full compatibility with ANSI FORTRAN 77; storage association between double-precision data and data of any other type does not.

You can override the default size of any variable with length override specifications in its declaration. You can also use intrinsic functions that are defined to take or return

INTEGER, REAL, COMPLEX, or LOGICAL values of a specific length.

This option takes advantage of increased REAL precision without paying the performance penalty of REAL*16 arithmetic for DOUBLE PRECISION variables.

-r*n*

Specifies that REAL variable declarations with unspecified lengths are to occupy *n* bytes of storage, where *n* can be 4 (the default) or 8. Storage association among REAL, COMPLEX, and DOUBLE-PRECISION data is not maintained as specified by the ANSI-77 standard.

Note

The -r*n* option is supported for compatibility with previous releases only and has been replaced by the -p8 and -pd8 options. The -r*n* option should not be used.

-re

Causes the compiler to generate reentrant code for recursive or, when the Application Compiler is used for compilation, parallel invocation of subprograms. This option makes it possible to call subroutines from parallel loops.

Each invocation of a subprogram has its own copy of unsaved local variables. Arguments are passed on the stack instead of by argument packets. COMMON block variables and saved and data-initialized variables still are shared among invocations.

If you compile a program using the -re option, all unsaved local variables are undefined on entry to the subprogram and must be assigned a value before being referenced.

The -re option is the default on CONVEX SPP Series machines.

-S

Generates symbolic assembly code for each program unit in a source file. The assembly code for source myfile.f is written to myfile.s. The assembly file is not assembled to produce object code. See the "Overview" section of this chapter for an overview of the Fortran compilation process.

-tm *target*

Specifies the target machine architecture for which compilation is to be performed. If you do not specify this compiler option, the compiler generates instructions for the class of machine on which it is running.

SPP Series only

On CONVEX SPP Series machines, *target* must be `spp1`. (Specifying `-tm spp1` is equivalent to the default behavior on SPP Series machines.)

C Series only

On C Series machines, *target* can be any one of the following values; an executable compiled for a particular architecture can also be run on all higher-series architectures:

target value	target architecture for compilation
`c1`	CONVEX C1 Series
`c2`	CONVEX C2 Series
`c32`	CONVEX C3200 Series
`c34`	CONVEX C3400 Series
`c38`	CONVEX C3800 Series
`c4`	CONVEX C4600 Series

On both C Series and SPP Series computers, if you specify a target machine, its instruction set is used regardless of the machine on which the compiler is running.

Note

The `file` utility may indicate that an executable generated with `-tm` specifying some C2 or C3 Series machine is a C1 executable. This will happen when no C2-only or C3-only instructions are used. Such an executable will run on a C1 machine, but it will contain instruction scheduling differences from a file generated for a C1 that `file` will not detect.

Debugging and profiling options

The options listed in this section allow CONVEX Fortran executables to be used with various program development tools.

`-al`

Noncharacter arrays declared with a last dimension of 1 are treated as if they were declared assumed-size (last dimension of `*`). Subscript checking can then be performed if the `-cs` option is also specified.

`-cs`

Compiles code to check that each subscript is within its array bounds. Does not check the bounds for arrays that are

dummy arguments for which the last dimension bound is specified as *.

`-cxdb`

Generates symbolic debugging information for CXdb, the CONVEX visual debugger. You can use CXdb to debug programs compiled without the `-cxdb` option, but symbolic debugging information will not be available. CXdb is capable of debugging optimized code and this option can be used with all levels of optimization. While the `-il` and `-is` options can be used with `-cxdb`, only the calling routine can be debugged; the inlined routines in question cannot be debugged.

When this option is included on the command line, a subdirectory called `.cxdb` is created in the current working directory that contains auxiliary debugging information. Refer to Chapter 2, "Program development tools," or to the *CONVEX CXdb User's Guide* for more information. CXdb is an optional product.

`-cxpa`

Instruments the compiled code so that the CONVEX performance analyzer (CXpa) can measure its performance at the routine level and the loop level. CXpa is an optional CONVEX product.

This option can be used with all levels of optimization and with the `-db` option. It should not be used with the CXpa block-level profiling option, `-cxpab`.

`-cxpab`

Instruments the compiled code so that the CONVEX performance analyzer (CXpa) can measure its performance at the block level. It is not compatible with the `-cxpa` option. The performance analyzer (CXpa) is an optional CONVEX product.

This option can be used with all levels of optimization and with the `-db` option.

`-cxpalib`

Instructs the compiler to link your program with the system installed libraries that are instrumented for use with CXpa. Programs profiled with `-cxpalib` may execute much slower. Any errors in the instrumentation of the libraries may cause the program to core dump. The amount of profiling data under this option is significantly increased.

`-cxpamon="dirname"`

Allows CXpa users to select a specific monitor instrumentation library by specifying the directory where the library resides. The full path of directory *dirname* is specified surrounded by quotation marks and indicates the directory path where the file cxpamon.o exists. The `-cxpamon` option facilitates using multiple versions of CXpa on a system.

`-cxpar`

Generates routine-level instrumentation for profiling under CXpa.

`-dc`

Specifies that a line with a `D` in column 1 is to be compiled and not treated as a comment line. Statements with a `D` in column 1 can be conditionally compiled, making this feature a useful debugging tool.

C Series only

`-metrics`

Writes the CXmetrics data file *sourcefile*`.met` in the same directory as the source file. *sourcefile* is the name of your original Fortran source file without the `.f` or `.FOR` extension.

sourcefile`.met` is a human-readable ASCII file used by CXmetrics to generate reports containing analytical data about the relative complexity of the program. Refer to the *CONVEX CXmetrics User's Guide* or to the metrics(1) man page for more information. CXmetrics is an optional product.

`-p`

Produces monitor routines for the `prof` profiler. The `prof` profiler is available for C Series machines as part of the optional CONVEX Consultant package.

C Series only

`-pb`

Produces information that the `bprof` profiler can use to generate source-level execution counts. The `bprof` profiler is part of the optional CONVEX Consultant package.

`-pg`

Produces information that the `gprof` profiler can use to generate a comprehensive execution profile. The `gprof` profiler is available for C Series machines as part of the optional CONVEX Consultant package.

-sc

Provides a syntax check. Stops compilation of each program unit in a source file after the program has been determined to be a valid Fortran program. Using this option during program development reduces compilation times.

Message and listing options

The options listed in this section provide control over program listings and advisory and warning messages. Cross-referencer options are also listed.

-errnames

Output the message name when a compile-time error message is issued. This name can then be used to look up the error in the list contained in Appendix C, "Compiler messages." This list contains more detailed explanations for many common compile-time messages, and, where appropriate, includes an example of code that can cause the error.

-LST

Generates a listing of the source program, inserts any compiler error messages at the appropriate points, and writes it to stdout. If the output is printed, the printer must be capable of printing 90 columns or more and 60 lines or more unless these values are adjusted using the -pl and -pw options; the printer must understand ASCII formfeeds.

-LSTI

Generates listing similar to -LST, but with all INCLUDE files expanded.

-na

Suppresses all advisory diagnostic messages.

-nw

Suppresses all warning diagnostic messages.

-or *table*

Specifies the contents of the optimization report; either the loop table, the array table, or both, can be displayed. The value for *table* can be all, none, loop, private, or array. If the -or option is not specified, only the loop table is displayed. For more information about the optimization report, see the "Optimization report" section in this chapter.

`-pl n`

Specifies a default maximum page length of *n* lines per page for the `-LST`, `-LSTI`, `-xr`, and `-xra` options. *n* does not include lines that are wrapped by the printer. If your printer wraps rather than truncates lines that exceed its carriage width, you may wish to supply a smaller *n* value to allow room for them. Default is *n* = 60.

`-pw n`

Specifies a desired header width of *n* columns per line for the `-LST`, `-LSTI`, `-xr`, and `-xra` options. This width only effects the listing's header; body lines that exceed *n* columns in width are not adjusted. If your printer wraps these lines, you may wish to adjust the page length setting (via the `-pl` option) to allow room for the extra lines. Default is *n* = 80.

`-xr`

Invokes the cross-referencer (`fcxref`) and generates a cross-reference report at compilation time.

`-xra`

Invokes the cross-referencer (`fcxref`) and generates a data file but does not generate a report. The report can be generated by running `fcxref`. Refer to Chapter 2, "Program development tools," for more information on `fcxref`.

The options listed in Table 5 can be used with `-xr` and `-xra`.

Table 5 Cross-referencer (`fcxref`) options

Option	Description
`-iw` *n*	Specifies the desired identifier width. Default is *n* = 16.
`-xrf` *file*	Uses *file* as the cross-referencer data file instead of `.fcxrefData`.
`-xrr` *file*	Sends the cross-reference report to *file* rather than to `stdout`.

Table 5 (continued) Cross-referencer (`fcxref`) options

Option	Description
`-xrg` *n*	Produces composite/global common block reports after all routine reports. Determine *n* by summing the desired report formats from the following list: 1: exhaustive members list, ordered by member offsets 2: exhaustive members list, ordered by member names 4: differing members report, ordered by member offsets 8: differing members report, ordered by member names If $n = 0$, no common block reports are generated; if $n = 15$, all 4 reports are produced. Default is $n = 8$.
`-xrm` *n*	Controls caller/callee matching reports. *n* takes one of the following values: 0: Omit caller/callee matching reports 1: Print call headers only 2: Print call mismatches only 3: Print both headers and mismatches If the `-xrm` *n* option is not specified, `fcxref` defaults to `-xrm 3`. For more information, refer to Chapter 2, "Program development tools."

C Series only

`-xro`

Calls the old `fxref` cross-reference generator. The following

options are related to this option:

Option	Description
`-iw` *n*	Specify the column width for identifiers. *n* can range from 8 to 32. The default is 16.
`-pw` *n*	Specify the logical page width used by the output formatter. The default is 132.
`-sl`	Produce a source listing with line numbers that precedes the cross-reference table. The `-LST` option is recommended rather than `-sl`.

C Series only

`-xrl`

Calls the `fxref` cross-reference generator and puts all objects (such as variables and arrays) into one table, rather than printing a separate table for each class of objects.

Note

The `-xro` and `-xrl` options are not supported for programs containing `REAL*16` data or Fortran 90 extensions.

Preprocessor options

The options listed in this section concern the use of preprocessors. Refer to Appendix I, "Preprocessor," for more information about the Fortran preprocessor.

`-fpp`

Runs the Fortran preprocessor as the first step of compilation. If this option is not specified, the preprocessor is not used. Refer to Appendix I for details on the use of this option.

`-D`*name*`[`*=def*`]`

Defines *name* to the preprocessor as if it had been specified in a `#define` statement. If no definition is given, the name is defined as 1.

`-E`

Runs only the Fortran preprocessor on the named Fortran programs and sends the result to the standard output file.

-I

The -I option can be used with or without the preprocessor. Refer to the "Miscellaneous options" section for more information.

-U*name*

Removes any initial definition of *name*. The only built-in names defined are "__LINE__", and "__FILE__".

-pp

Invokes a user-defined preprocessor. Refer to Appendix I, "Preprocessor."

Miscellaneous options

The options listed in this section are not otherwise categorizable.

-align cseries

Causes the compiler to store COMMON blocks using "tight" packing (the ANSI standard). Tight COMMON block packing is the default on CONVEX C Series machines.

This method does not pad COMMON block data items to boundaries, so partial-word items may cause other data items to align on unnatural boundaries. For instance, a REAL*8 item may align on a 4- or 6-byte boundary instead of an 8-byte boundary.

For more information, refer to the "COMMON block packing" section of *Fortran Language Reference* Chapter 5.

-align spp

Causes the compiler to align all COMMON block data items to their natural boundaries (up to 8 bytes). This is the default on SPP Series machines. For example, REAL*4 items are padded to 4-byte boundaries, REAL*8 items to 8-byte boundaries, and so on.

This can improve performance in cases where partial-word items appear in COMMON blocks. For more information, refer to the "COMMON block packing" section of *Fortran Language Reference* Chapter 5.

Note

Standard-conforming programs that use non-identical COMMON block declarations in different subprograms to achieve an effect similar to EQUIVALENCE may not work correctly when compiled with the -align spp option.

C Series only

`-ansic`

Links `/usr/lib/libc.a` to your program. `libc.a` is the extended POSIX library, which includes the ANSI C library. This link allows mixed ANSI C and Fortran programs. `-ansic` is the default; this option is provided to override a `-pcc` specified in `FCOPTIONS`. When `libc.a` is loaded, many library aborts produce signal #6 (`SIGIOT`) instead of signal #4 (`SIGILL`) and print "`IOT`" instead of "`Illegal Instruction`" when they abort.

`-B`*dir*

Finds the substitute compiler (`fskel`, `fpp`, and `errmsgf`) in the directory named *dir*.

Normally on CONVEX SPP Series machines, the current path of `fc` is used as the path from which related executables and libraries are searched for.

`-I`*dir*

For `'INCLUDE'` files (or, if used with the preprocessor, `#include`) whose names do not begin with "/", the compiler (or preprocessor) searches first in the directory of the file argument, then in directories named in `-I`. No more than eight directories can be specified. If used with the `-fpp` option, this option is passed to the preprocessor.

`-link` *arg*

Passes *arg* to the loader where *arg* is an arbitrary loader option. You must delimit *arg* within quotation marks if a blank space appears in the argument.

A library linked with the `fc` driver cannot contain a C main program.

`-nosc`

Disables short circuit evaluation of conditionals. See Chapter 8 of the *Fortran Language Reference* for more information.

SPP Series only

`-noU77`

Prevents a trailing underscore character (_) from being added to names of routines in the `libU77` Fortran library (or any variety of `libU77`, such as `libU77p8`). 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.

`-o` *name*

Assigns *name* as the name of the executable file produced by the loader. The default name is `a.out`. If the loader is not invoked because the `-c` option is specified and if there is only one file to compile or assemble, then *name* becomes the name of the object module.

C Series only

`-pcc`

Links a Portable C Compiler compatible `libc.a` to your program, allowing you to mix only the `pcc` dialect of C with Fortran programs. This option overrides the default `-ansic` option, which allows you to mix ANSI C and Fortran programs. This option has no effect on SPP Series machines.

SPP Series only

`-ppu`

Causes 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.

C Series only

`-tl` *n*

Sets the maximum CPU time limit for compilation to *n* minutes. If the time limit is exceeded, compilation terminates with the message "`System error in /usr/convex/fskel`."

`-vn`

Display information concerning the version of the compiler that is being used. Output goes to `stderr`.

`-W`*subproc,sparg* `[,`*sparg*`...]`

Passes specified argument(s) to subprocess *subproc*. Each argument (*sparg*) takes the form `-`*arg*`[,` *argvalue*`]`, where *arg* is the name of an option recognized by the subprocess and *argvalue* is a separate argument that is included if required.

The `-W` option specification allows additional, implementation-specific options to be recognized by the compiler driver.

The following values are recognized for *subproc*:

p	preprocessor
c	compiler
0	compiler
a	assembler
l	linker

For example,

```
-Wl,-a,archive
```

causes the linker to link with the archive libraries.

`-72`

Processes only the first 72 characters of each program line (the compiler normally processes all characters). A line with fewer than 72 characters is padded with blanks until 72 characters are processed. A tab counts as one character. The `fsplit` utility ignores characters beyond column 72 when this option is specified, but it does not remove them from the file. See the fsplit(1F) man page for more information.

Loading programs

After a program has been compiled, the loader (`ld`) must process the resulting object code before it can be executed. By default when you compile using `fc`, `ld` combines object files as needed to produce an executable program, as illustrated in Figure 1 on page 2.

The loader performs the following functions:

- Combines your program object code with object code from libraries
- Resolves external references to functions, subroutines, and common blocks
- Creates an executable module

The default name of the executable module is `a.out` unless you specify a different name with the `-o` *name* compiler option. You can invoke the loader by using either the compiler command (`fc`) or the loader command (`ld`).

Using the `fc` command

It is strongly recommended that you invoke the loader with the `fc` command, using the most current release of the CONVEX Fortran compiler. This approach ensures that the proper Fortran

libraries are automatically loaded in the proper order. The compiler, in turn, passes any loader options on the `fc` command line to the loader.

On both C Series and SPP Series computers, you can use the `-W` compiler option to pass loader options to the loader. You can, but need not, perform compilation when using the `fc` command to invoke the loader. You can suppress the loading phase of the compilation with the `-c` option.

Executing programs

To execute your program after the loader has processed it, enter the name of the executable file. The default name for the executable file is `a.out`. If you have included the `-o` *name* option on the `fc` command line, the name of the executable file is *name*.

Messages

This section presents an overview of the messages that can appear during compilation and at runtime. The messages are grouped into the following categories:

- Compiler messages
- Runtime error messages
- Optimization report

Compiler messages

The compiler may produce various error, warning, and informational messages during compilation. These messages are directed to the standard error file (`stderr`). On C Series machines, `stderr` is unit 0; on SPP Series machines it is unit 7.

A compiler message includes the line and column number of the text in which the error occurs, the path name of the source file containing the text in error, and a brief description of the error.

Examples:

```
fc: Error on line 7.1 of testprog.f: Label
defined but never referenced.

fc: Warning on line 3.4 of myprog.f: Divide
by zero may occur at runtime.
```

The number before the decimal point is the line number where the error occurs. The number following the decimal point is the column number.

If an internal compiler error occurs, the compiler outputs a message that begins with the words `COMPILER ERROR`. Such a message should be reported to the CONVEX Technical Assistance Center (TAC).

For more information on compiler messages, refer to Appendix C, "Compiler messages."

Runtime error messages

Runtime error messages, which can be generated by math routines, I/O operations, or trap errors, are directed to the standard error file (`stderr`). I/O error messages can contain up to four lines of information, depending on the operation involved.

The following examples show typical runtime error messages. The numbers in brackets indicate an error number associated with a particular error condition.

Examples:

```
mth$r_sqrt: [300] square root undefined for negative
numbers

mth$r_sqrt: [300] square root undefined for negative
values
sqrt( -1.7014117E+38)= 1.3043818E+19

write sfe: [100] error in format
logical unit 6, named 'stdout'
lately:writing sequential formatted external IO
part of last runtime format: (f6.2,x,v5|,x,i

dofio: [115] read unexpected character
logical unit 7, named 'fort.7'
lately: reading sequential formatted external IO

part of last pre-compiled format: (14,F7.2,E10.4|,E10
```

Some runtime errors also produce a stack trace.

Refer to Appendix D, "Runtime messages" for a list of error numbers and explanations.

Optimization report

If a program is compiled with the `-O2` or `-O3` option, the compiler generates an optimization report for each program unit. This report contains a loop table, a privatization table, an array table, or all three.

You can specify which tables are to be included in the optimization report, or suppress generation of the report, with the `-or` compiler option. Refer to the *Fortran Optimization Guide* or the *Exemplar Programming Guide* for a complete description of the optimization report. A summary description follows.

Loop table

The loop table lists the optimizations that were performed on each loop. On SPP Series machines, this table lists the final order of nested loops after any interchange has occurred

The analysis, test, and variable name footnote tables are included if necessary to supplement the loop table.

Analysis table
If necessary, an analysis table is included in the optimization report to provide details on optimizations reported in the loop table and, if appropriate, the reasons why a possible optimization was not performed.

Test table
If any test promotion or removal optimizations were performed, a test table is included in the optimization report.

Variable name footnote table
Variable names that are too long to fit in the `Iter. Var.` columns of the loop and array table sections are truncated and followed by a colon and a footnote number. These footnotes are expanded in the variable name footnote table.

Privatization table

The privatization table lists which user variables were privatized for parallelization and the type of privatization that occurred for each.

Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed.

Program interfaces

Calling sequences for the runtime system include the use of the short-form assembly language call instruction (`callq`) and additional conventions related to the use of registers and the values in registers after calls. The compiler provides versions of

the intrinsics that use the standard calling sequence. This feature allows you to pass intrinsic names as arguments to subprograms.

There is a performance penalty for invoking intrinsics passed as arguments.

The runtime system provides scalar versions of the math intrinsics. On C Series machines, vector versions of these intrinsics also are available, and the compiler determines which version of the routine to call.

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 6, where *name* represents the symbolic name.

Table 6 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*

Chapter 4 describes CONVEX Fortran's naming conventions in more detail. In addition to coverage of the `-noU77` and `-ppu` naming options (both available on SPP Series machines only), the chapter contains several interlanguage programming examples that illustrate how to reference external names from both Fortran and C programs.

Program development tools

2

Development tools

This chapter presents an overview of the tools available for debugging and analyzing Fortran programs. These tools consist of:

- Cross-reference generator (`fcxref`)
- Assembly-language debugger (`adb`)
- CONVEX Visual Debugger (CXdb)
- Performance analyzer (CXpa)
- Profilers
- CXtrace (SPP Series only)
- Consultant III and CXtools
- CONVEX Application Compiler
- Postmortem dump (C Series only)
- `error` utility

Some of these tools are distributed with the compiler; others are optional CONVEX products.

Cross-reference generator

The cross-reference generator (`fcxref`) produces a detailed report of references to each object in a Fortran source program. Objects cross-referenced in the report include:

- Variables
- Arrays
- Parameters
- Labels
- Constants
- Modules

- Functions
- Subroutines
- Intrinsic routines
- COMMON block names
- Include files
- Cray pointers
- VAX structures and unions

Cross-reference information is generated through use of either the -xr or the -xra command line option. Both options generate a data file that contains all the cross-referencing information needed to build a cross-reference report. When the -xr option is used, the data file is deleted once the report is created. The default name for the data file is .fcxrefData; this name can be changed through use of the -xrf command line option described in Table 7.

The -xr and -xra options differ in that -xr can independently cross-reference several source files; -xra can globally cross-reference several Fortran source files.

The -xr option generates both the data file and the report at compile time and (by default) sends the report to stdout along with any other fc output; the data file is then deleted. The -xra option appends cross-referencing information to the data file on each compilation in which it is specified but does not delete the data file or produce the actual report. After compiling all the component programs or routines with the -xra option, you can use the fcxref command to generate the report.

Cross-referencer options

Command line options associated with the cross-referencer are listed in Table 7. fcxref options can be specified in the FCOPTIONS environment variable, where they are automatically used on invocation of the compiler or cross-referencer.

Table 7 Cross-referencer options

Option	Command line(s)	Description
-xr	fc -xr	Invokes cross-referencer and generates report at compile time.
-xra	fc -xra	Invokes cross-referencer and generates data file but does not generate report. Useful for globally cross-referencing several separate Fortran programs.
-xrf *file*	fc -xr -xrf *file* fc -xra -xrf *file* fcxref -xrf *file*	Uses *file* as the cross-referencer data file instead of .fcxrefData. This is useful if you are using the -xra option to generate several different cross-reference reports with several different groups of programs. This option is of little use when used with -xr because the data file is automatically deleted after compilation.
-xrd	fcxref -xrd	Deletes the data file after generating the report. By default, the data file is saved when fcxref is run standalone.
-xrr *file*	fc -xr -xrr *file* fcxref -xrr *file*	Sends the cross-reference report to *file* rather than to standard output.
-xrg *n*	fc -xr -xrg *n* fcxref -xrg *n*	Produces composite/global COMMON block reports after all routine reports. Determine *n* by summing the desired report formats from the following list: 1: exhaustive members list, ordered by member offsets 2: exhaustive members list, ordered by member names 4: differing members report, ordered by member offsets 8: differing members report, ordered by member names If *n* = 0, no COMMON block reports are generated; if *n* = 15, all 4 reports are produced. Default is *n* = 8. For more information, refer to the "Cross referencer report" section of this chapter.
-xrm *n*	fc -xr -xrm *n* fcxref -xrm *n*	Controls module interface reports. *n* takes one of the following values: 0: Don't print module interface reports 1: Print call headers only 2: Print call mismatches only 3: Print both headers and mismatches If the -xrm *n* option is not specified, fcxref defaults to -xrm 3. For more information, refer to the "Cross referencer report" section of this chapter.

Development tools

Table 7 (continued) Cross-referencer options

Option	Command line(s)	Description
`-iw` *n*	`fc -xr -iw` *n* `fcxref -iw` *n*	Specifies the desired identifier width. This is the width of the fields in which symbol names are printed for each routine in the report. Shorter symbols are padded with spaces; longer symbols are truncated on the right. Default is *n* = 16. Programs using VMS structures may require larger values of *n*.
`-pl` *n*	`fc -pl` *n* `fc -xr -pl` *n* `fcxref -pl` *n*	Specifies a default maximum page length of *n* lines per page for the `-xr` and `-xra` options as well as the `-LST` and `-LSTI` options. *n* does not include lines that are wrapped by the printer. If your printer wraps rather than truncates lines that exceed its carriage width, you may wish to supply a smaller *n* value to allow room for them. When *n*-1 lines have been printed in the report, an ASCII form feed followed by a page header is automatically inserted (one line per page is reserved for the form feed character). Default is *n* = 60.
`-pw` *n*	`fc -pw` *n* `fc -xr -pw` *n* `fcxref -pw` *n*	Specifies a desired page width of *n* columns per line for the `-xr` and `-xra` options as well as the `-LST` and `-LSTI` options. This width only effects the listing's header; body lines that exceed *n* columns in width are not adjusted. If your printer wraps these lines, you may wish to adjust the page length setting (via the `-pl` option) to allow room for the extra lines. Default is *n* = 80.

Note

The `fxref -sl` source listing option is no longer supported. Use `-LST` compiler option instead; refer to Chapter 1.

Cross-referencing with `-xr`

Use the `-xr` option on the `fc` command line to generate separate cross-reference reports for individual Fortran source files at compile time. `-xr` creates a cross-reference data file named `.fcxrefData` by default and deletes this file as soon as compilation is completed. This file name can be changed by using the `-xrf` option (refer to Table 7).

```
fc progl.f -xr -xrg 10 -xrr progl.xr -pw 78 -iw 12
```

This line compiles the program `progl.f` and generates a cross-reference report that includes COMMON block reports using the exhaustive members list and differing members report formats, both ordered by member names. The report is stored in

the file progl.xr, and is formatted on a 78-column-wide page with 12-column symbol-name fields.

You can generate separate cross-reference reports for individual source files compiled on the same fc command line with -xr also. In this case, cross-reference reports are generated for each file during compilation and sent to stdout (or to the -xrr specified file) under separate cover pages, separated by ASCII form feeds.

```
fc progl.f prog2.f prog3.f -xr -xrr progs.xr
```

This line compiles progl.f, prog2.f and prog3.f, generates a cross-reference report for each one (using default report option values), and places all the reports in the file progs.xr. Note that although they are placed in the same output file, the reports generated are *independent*; no cross-referencing is done between input files.

Cross-referencing with -xra

Specifying -xra on the fc command line generates the cross-reference data file but not the report. -xra appends to the cross-referencer data file rather than overwriting it. This option leaves the data file intact when compilation and/or report generation is complete; it can then be further appended to by compiling additional programs with -xra. Redundant compilations of the same source file with -xra will delete data generated by previous compilations of that source file and replace it with data from the most current compilation.

The -xra option is useful for generating a global cross-reference across several separate Fortran source files. Objects from each separate file are combined and sorted together; the files are reported on as a unit, exactly as if they were all contained in one source file.

```
fc progl.f -xra -xrf progl.xrdata
```

Here the -xrf option specifies a cross-referencer data file name; when -xra is specified on the fc command line, -xrf is the only other cross-referencer option allowed. After compiling all the desired programs, generate the report by running fcxref with any desired options:

```
fcxref -xrf progl.xrdata -xrg 10 -iw 12 -pw 78 -xr progn.xr
```

The above command line generates a cross-reference report based on the data contained in the file progl.xrdata. The

report includes `COMMON` block reports using the exhaustive members list and differing members report formats, both ordered by member names. It is formatted with 12-character identifier fields and a 78-column page, and is stored in the file `progn.xr`. Because the `-xrd` option is not used, the data file is left intact after report generation.

If the `-xrf` option is omitted from the `fc` command line, `fcxref`, by default, uses `.fcxrefData` as its data file; no `-xrf` option or data file name would be necessary on the `fcxref` command line. It is important to remember that when the `-xrf` option is used, the data file it creates is appended to with each compilation (unless `-xrf` is used to switch data files) and is *not* automatically deleted. Data files can therefore become quite large. To conserve disk space, be sure to manually remove these data files when you are through with them, or use the `-xrd` cross-referencer option to automatically remove them.

All cross-referencer options that can be used on the `fc` command line with the `-xr` option can be used with `fcxref`. The `-xrd` (delete data file after report generation) option is also available to `fcxref`.

The `-xra` option can be useful in makefiles that compile a number of source files that you wish to cross-reference later.

Cross-reference report

Each section of the cross-reference report is explained below, along with examples of cross-reference report segments.

Cover page

The cover page lists the date and time at which the report was generated followed by the report parameter values, which include identifier width, page width, page length, data file name, report file name, level of common reports, level of module reports, whether to delete the data file, and whether `fcxref` is being run standalone or by the `fc` driver. See Figure 2.

Figure 2 Cross-referencer report cover page

```
CONVEX Fortran CROSS-REFERENCE REPORT
-------------------------------------


DATE => Thursday, April 25, 1991
TIME => 06:52:11  p.m.


REPORT PARAMETERS:
identifier width => 16
page width       => 80
page length      => 60
data file name   => .fcxrefData
report file name => test.xr
common reports   => 8
module reports   => 3
delete data file => yes
stand alone run  => no
```

Development tools

Routine reports

Routine reports generally constitute the largest part of the report. A routine report is generated for each routine in the cross-reference report; they are ordered alphabetically by routine name. Routine reports list the routine name, type (subroutine, function, or main) and the source file it resides in, followed by symbol, structure (if VMS structures are used) and COMMON block summary reports. For each routine's symbol summary, objects in the routine are listed alphabetically in the left column. In the structure and COMMON block summaries, objects are ordered by increasing offsets. Object types are presented as follows:

- Objects preceded by dollar signs ($) are VMS structures. These may be anonymous, in which case the cross-referencer labels the *n*th structure `<ANONYMOUS#n>`. VMS unions and maps are always anonymous. They are listed as such in the structure summary report; however, when fields in unions are presented in the routine report, union and map information is removed, and the field is qualified only by the name of the enclosing structure. Dots (.) separate path elements in record variable accesses and structure fields.
- Objects preceded by underscores (_) are COMMON blocks.

- Other objects are labels, variables, parameters, block data names or routine names.

Symbol summary

The symbol summary is the first part of the routine report. It lists and briefly describes every object in the routine.

The right column of the symbol summary contains the specific object type (variable, array, parameter, label, main program, subroutine, function, intrinsic function, block data, COMMON block, Cray pointer, or VAX record), and relevant additional information, such as variable or function type, array bounds, offsetting information for record members, COMMON block name if the object resides in a COMMON block, etc. For generic intrinsic functions, the type reported is the default type for the function; the actual intrinsic type is based on the argument type and may not correspond to this.

The right column also contains line number references in the following format:

linum [*inclindx*] [*contextop*]

linum indicates the line number in the source file and may not correspond to line numbers generated in -LST listings. Every object will have at least one line-number entry because it must appear at least once in the program in order to be included in the cross-reference report; the other line number reference information is optional.

The *inclindx* field is present only if the object is contained in an include file; this is the index to the include table that appears before the table of contents near the end of the report (see description below).

The *contextop* field contains a context operator that indicates what is done with the object at the given line. Context operators are described in Table 8.

Table 8 fcxref context operators

Operator	Description
<blank>	Object referenced.
#	Main program, subroutine, function or COMMON block name appearance in header line.
=	Object assigned or defined; denotes assignment statements (=), input statements (READ, ACCEPT), data statements, statement label assignments (ASSIGN), appearance of variable as dummy or actual subroutine argument.
*	Object described; denotes declaration statements (symbol declarations), dimension statements (for array names), static and automatic storage class statements, external statements, intrinsic statements, program or format label definition line, COMMON block names in common statements, pointers and pointees in Cray pointer statements, statement function declarations, and VAX structure, field, union, map and record declarations.

A cross-referencer output example illustrating how various objects are handled in the routine report is shown in Figure 3. This example, along with the examples in Figure 4 through Figure 10 were compiled with the following command line:

```
fc test.f -c -vfc -O3 -xr -xrr test.xr
```

Thus the cross-referencer output is stored in a file named test.xr.

Figure 3 Cross-referencer routine report

```
CONVEX Fortran CROSS-REFERENCE                        AGE: 13
ROUTINE: MAIN
*****************************************************************
ROUTINE: MAIN
KIND:    MAIN-PROGRAM
FILE:    test.f
*****************************************************************

$BIRTH            TYPE-NAME  RECORD
                  8*  173  174  178

$BIRTH.CITY       RECORD-FIELD  CHARACTER*10  OFFSET=>12
                  10*  190
                  .
                  .
 5                 LABEL
                  18*  26

A                 VARIABLE  REAL*4  COMMON-MEMBER BLOCK=>_BLK1
                  50  56=  56  57

B                 VARIABLE  REAL*4  COMMON-MEMBER  BLOCK=>_BLK2
                  3[4] 4[4]=

EMPL              VARIABLE  $BIRTH*12  RECORD
                  173*

EMPL.CITY         VARIABLE  CHARACTER*10  PATH-NAME  OFFSET=>0
                  186

FUNC1             FUNCTION  REAL*4
                  22
                  .
                  .
IARR              VARIABLE  INTEGER*4  STATIC ARRAY(1:5)
                  11*  25=
                  .
                  .
MAIN              MAIN-PROGRAM
                  3#
                  .
                  .
_BLK1             COMMON-BLOCK
                  5*
                  .
                  .
                  .
```

Figure 3 shows the symbol summary section of the routine report for the main program, which is called MAIN. The header tells us it is included in the source file test.f. Object descriptions follow the header information.

$BIRTH is a type name for a record that is declared at line 8 and referenced on lines 173, 174, and 178. $BIRTH.CITY is a 10-character record field, offset 12 bytes from the beginning of the record, declared on line 10, and referenced on line 190. The symbol summary entry for 5 tells us that it is a LABEL symbol, that it is declared (in this case, it first appears) on line 18, and that it is again referenced on line 26. The entry for A tells us that it is a VARIABLE of type REAL*4, and that it is a member of the COMMON block _BLK1. A is referenced on line 50, assigned a value on line 56, referenced again on line 56 and also on line 57. Similarly, B is of type REAL*4 and a member of _BLK2. It is referenced on line 3 of include file number 4 (refer to the "Include file reference table" section of this chapter), and assigned a value on line 4 of this include file.

EMPL is a record of type $BIRTH*12, declared on line 173; EMPL.CITY is a variable (the CITY field of EMPL) of type CHARACTER*10, offset 0 bytes from the beginning of the record, and appearing on line 186.

FUNC1 is identified as a function of type REAL*4, and is referenced on line 22. IARR is a 5-element STATIC ARRAY of type INTEGER*4; we can infer from 11* that it is dimensioned at line 11 (because line 11 is the only reference of the variable, it must be where it is dimensioned), and some element of IARR is assigned a value at line 25. The entry for MAIN, the main program name, appears on line 3. The last entry in the report tells us that the COMMON block BLK1 is declared on line 5.

Structure summary

Figure 4 shows the structure summary section of the routine report.

Figure 4 Cross-referencer routine report structure summary

```
*****************************************************************

ROUTINE STRUCTURE SUMMARY

*****************************************************************


STRUCTURE => $<ANONYMOUS#0>
SIZE      => 4

OFFSET          FIELD
------          -----
0               $<ANONYMOUS#0>.J
                .
                .
                .

STRUCTURE => $BIRTH
SIZE      => 12

OFFSET          FIELD
------          -----
0               $BIRTH.CITY
10              $BIRTH.STATE
                .
                .
                .
```

In Figure 4, `$<ANONYMOUS#0>` on the first line refers to the 0th unnamed VMS structure in the file, which has a size of 4 bytes and a field called `J` which is offset 0 bytes from the start of the structure. `$BIRTH` is the other structure cross-referenced here; it is 12 bytes in size and contains 2 fields, `CITY` and `STATE`, with offsets of 0 and 10 respectively.

COMMON block summary

Figure 5 shows the routine COMMON block summary section of the routine report.

Figure 5 Cross-referencer routine COMMON block summary

```
CONVEX Fortran CROSS-REFERENCE
PAGE: 15
ROUTINE: MAIN

*********************************************

ROUTINE COMMON BLOCK SUMMARY

*********************************************

BLOCK = _BLK1
SIZE  = 12

OFFSET                MEMBER
------                ------
0                     A
4                     B
```

Figure 5 shows the routine COMMON block summary for BLK1. This summary tells us that the block is 12 bytes in size and contains as members A and B at offsets of 0 and 4 bytes respectively.

Module interface reports

The MODULE INTERFACES section of the report provides information on each module's location, type, and arguments. The -xrm *n* option allows you to specify the desired module interface information according to the values of *n* specified in Table 9.

Table 9 `-xrm` *n* options

n	Report
0	No module information
1	Show module call header information
2	Show module call mismatch information
3	Show both header and mismatch information

In absence of the `-xrm` option, the module interface report contains both header and mismatch information by default (`-xrm 3`). To exclude module interface reports you must specify `-xrm 0` when you generate the cross-reference report.

Specifying `-xrm 1` generates a `MODULE INTERFACES` section, which lists each module along with its location (line number and filename), type (function, subroutine or main program), number of arguments, and argument types. Specifying `-xrm 2` generates a `MODULE INTERFACES/CALL-POINT MATCHING` section, which, like the `MODULE INTERFACES` section, lists basic information on each module, but which also includes call cross-reference information and references into a table of error messages that is printed at the end of the module interface report.

Default module interface reports were generated for `test.f`, so both a `MODULE INTERFACES` section and a `MODULE INTERFACES/CALL-POINT MATCHING` section are included in the module interface report. Figure 6 shows part of the `MODULE INTERFACES/CALL-POINT MATCHING` section of the report. This section includes all the module header information (`LINE/FILE`, `KIND/TYPE`, and `ARGUMENTS`), plus call information, error information and an error message table.

Figure 6 Module interface section example

```
CONVEX Fortran CROSS-REFERENCE                                    PAGE: 28
****************************************************************************
MODULE INTERFACES / CALL-POINT MATCHING
****************************************************************************
FUNC1                LINE/FILE => 157, test.f
                     KIND/TYPE => FUNCTION REAL*8
                     ARGUMENTS => 3
                     FORMAL  1 => VARIABLE REAL*4
                     FORMAL  2 => VARIABLE LOGICAL*8
                     FORMAL  3 => VARIABLE REAL*4

                     *MISMATCHED CALL POINT
                     CALLER    => AAA
                     LINE/FILE => 229, test.f
                     KIND/TYPE => FUNCTION CHARACTER*5 [2]
                     ARGUMENTS => 2 [3]
                     ACTUAL  2 => VARIABLE REAL*4 [5]

RRR                  LINE/FILE => 205, test.f
                     KIND/TYPE => SUBROUTINE
                     ARGUMENTS => 0
                     *MODULE NOT CALLED FROM ANY CROSS-REFERENCED MODULE

SUB04                LINE/FILE => 80, test.f
                     KIND/TYPE => SUBROUTINE
                     ARGUMENTS => 1
                     FORMAL  1 => VARIABLE INTEGER*4
                     *ALL CALLS MATCH THIS MODULE INTERFACE

CALL-POINT MISMATCH ERROR MESSAGE TABLE:

[1]  CALL-POINT KIND DISAGREES WITH MODULE KIND
[2]  RETURN TYPE/PRECISION DIFFERENT THAN MODULE TYPE
[3]  WRONG NUMBER ARGUMENTS AT CALL-POINT
[4]  ACTUAL/FORMAL ARGUMENT KINDS ARE INCOMPATIBLE
[5]  ACTUAL ARGUMENT BASE TYPE/PRECISION DOES NOT MATCH FORMAL
[6]  SCALAR/ARRAY ARGUMENT CONFLICT
[7]  ARRAY ARGUMENTS HAVE DIFFERENT TOTAL NUMBER CELLS
[8]  READ-ONLY ARGUMENT IS ASSIGNED A VALUE IN MODULE
```

As shown in Figure 6, the left column of the module interface report contains the module name; the right column contains information pertaining to that module. According to the report,

the function FUNC1, which begins on line 157 of the file test.f, is of type REAL*8, and has 3 formal arguments: one REAL*4, one LOGICAL*8, and one REAL*4. The message *MISMATCHED CALL POINT signifies that FUNC1 is incorrectly called by module AAA at line 229 of test.f. The bracketed numbers ([2], [3] and [5]) that appear at the ends of the following lines under FUNC1 correspond to the CALL-POINT MISMATCH ERROR MESSAGE TABLE at the bottom of Figure 6. According to these references, the call is incorrect in that AAA is attempting to assign the result of FUNC1 to a CHARACTER*5 variable, it is passing an incorrect number of arguments to FUNC1, and the second actual argument does not match the type or precision of its corresponding formal argument in FUNC1.

RRR, the next module in the report, begins at line 205 of test.f, is a subroutine and has no arguments. This module is not called from any module included in this cross-reference report.

SUB04, the last module included in this sample report, begins on line 80 of test.f, and is a subroutine with one argument. All calls to SUB04 are correct.

The CALL POINT MISMATCH ERROR MESSAGE TABLE, which appears at the end of the section, describes call-point mismatch errors referenced by bracketed numbers in the various modules. The error message are described in more detail below.

[1] CALL-POINT KIND DISAGREES WITH MODULE KIND

A function was called as a subroutine, a subroutine was called as a function, or a main program was called.

[2] RETURN TYPE/PRECISION DIFFERENT THAN MODULE TYPE

The type or precision returned by a function differs from the function's declared type or precision. For example, a function declared as type INTEGER in the main program is declared as type REAL in its function statement.

[3] WRONG NUMBER ARGUMENTS AT CALL-POINT

A module was called with the incorrect number of arguments.

[4] ACTUAL/FORMAL ARGUMENT KINDS ARE INCOMPATIBLE

The actual argument is of a different kind than the formal argument; for example, an attempt was made to pass a label into a variable or vice versa.

[5] ACTUAL ARGUMENT BASE TYPE/PRECISION DOES NOT MATCH FORMAL

The type or precision of the actual argument differs from the type or precision of the formal argument. For example, an attempt was made to pass an actual argument of type REAL*8 where a formal argument of type REAL*4 is expected.

[6] SCALAR/ARRAY ARGUMENT CONFLICT

A scalar was passed where an array is required or vice versa.

[7] ARRAY ARGUMENTS HAVE DIFFERENT TOTAL NUMBER CELLS

The total number of elements in an array argument differs from the array size declared in the receiving module. Formal and actual array arguments can vary in shape but must have the same number of elements.

[8] READ-ONLY ARGUMENT IS ASSIGNED A VALUE IN MODULE

The module is attempting to modify an actual argument that is a constant value or symbolic name.

Caller/callee routine cross-reference

The CALLER/CALLEE ROUTINE CROSS-REFERENCE section of the report (see Figure 7) lists each routine's subroutine, function or intrinsic callers and callees.

Figure 7 Caller/callee routine cross reference example

```
CONVEX Fortran CROSS-REFERENCE                                    PAGE:28

*****************************************************************

CALLER/CALLEE ROUTINE CROSS-REFERENCE

*****************************************************************
                 .

FUNC1            KIND => FUNCTION
                 FILE => test.f
                 CALLS.......ENTRY, PPENTRY2
                 CALLED-BY...MAIN
                 .
                 .
                 .
MAIN             KIND => MAIN-PROGRAM
                 FILE => test.f
                 CALLS.......FUNC1, SIN, SQRT, SUB1

SUB1             KIND => SUBROUTINE
                 FILE => test.f
                 CALLS.......FUNC2, SUB3
                 CALLED-BY...FUNCX, MAIN, SUB2
                 .
                 .
                 .
```

Figure 7 is fairly self-explanatory. The object name is listed in the left column. In the right column, the `KIND =>` entry indicates whether the object is a main program, subroutine, function, or intrinsic routine; `FILE` indicates the file that contains the object; `CALLS` and `CALLED-BY` indicate routines that the object calls and is called by, respectively.

Composite COMMON block reports

For each COMMON block, this section of the report lists the block name and maximum size, followed by information concerning each member's name, offset and declaring routine. This information can be ordered and formatted with the `-xrg` option; see Table 7. The following formats are available:

1. Exhaustive members list, ordered by member offsets: this format lists all members of all COMMON blocks declared

among all routines in the program, ordering the list by member offsets.

2. Exhaustive members list, ordered by member names: produces the same exhaustive list as in item 1, but orders it by member names.
3. Differing members report, ordered by member offsets: similar to item 1, but all occurrences of a member offset are collapsed onto one report line. Routine names that do not fit on the report line for a given member are truncated. Useful in identifying cases where a given offset is host to different members.
4. Differing members report, ordered by member names: similar to item 2, but all occurrences of a member name are collapsed onto one report line. As in item 3, routine names that do not fit on the report line for a given member are truncated. Useful in identifying cases where a member is declared at different offsets by different routines.

Figure 8 shows the differing members report for BLK1 ordered by member names.

Figure 8 Cross-referencer COMMON block summary example

```
CONVEX Fortran CROSS-REFERENCE                              PAGE: 31

*****************************************************************

COMPOSITE COMMON BLOCK SUMMARY: DIFFERING MEMBERS

*****************************************************************

BLOCK -> _BLK1
SIZE  -> 412  [smaller sizes were observed]

MEMBER           OFFSET           ROUTINES
------           ------           --------
A                0                MAIN
B                0                SUB1
B                4                MAIN
C                4                SUB1
C                8                MAIN
                 .
                 .
                 .
```

Development tools

The SIZE => entry in Figure 8 shows that the COMMON block is 412 bytes in its largest definition. If any subroutines declared the block with a size less than 412 bytes, as is the case here, the message [smaller sizes were observed] is appended to the size. The appearance of this message may help you find overlapping COMMON blocks, which can cause bugs.

The remaining entries to this summary are self-explanatory.

Include file reference table

The `INCLUDE FILE REFERENCE TABLE` section of the report (see Figure 9) presents a numbered list of include files referenced.

Figure 9 Include file reference table example

```
CONVEX Fortran CROSS-REFERENCE                                    PAGE: 33

****************************************************************************

INCLUDE FILE REFERENCE TABLE

****************************************************************************

1)   ./incl0.f -of- test.f
2)   ./incl1.f -of- test.f
3)   ./incl2.f -of- test.f
4)   ./incl4.f -of- test.f
```

The numbers in the left column of Figure 9 are include table indexes. These are the bracketed numbers referred to in the routine reports by objects that are contained in include files.

Table of contents

The table of contents (see Figure 10) lists subroutines, functions and COMMON blocks contained in the report and the report page numbers on which they appear, ordered by object name.

Figure 10 Cross-referencer table of contents example

```
CONVEX Fortran CROSS-REFERENCE                                   PAGE:34

************************************************************************

TABLE OF CONTENTS

************************************************************************
                   .
                   .
                   .
MAIN               MAIN-PROGRAM.........13
                   .
                   .
                   .
_BLK1              COMMON-BLOCK.........4, 11, 14, 17, 20
                   .
                   .
                   .
```

Object names are listed in the left column. The right column shows the object type and cross-reference report page number on which it can be found.

Files and required utilities

`fcxref` must have access to the `sort` utility program. Make sure `sort` resides along a path contained in your path shell variable.

`fcxref` stores work files in `/tmp` while it executes; these files are automatically deleted on normal completion of the cross-referencer run.

Assembly-language debugger

The assembly-language debugger (`adb`) is an object-code debugger that requires no recompilation or special compiler options. Use `adb` to examine core dumps from failed programs and to interactively debug programs at the assembly-language level.

Because `adb` runs programs under its control, it is always aware of the state of the program and the values of all variables. Using `adb`, you can:

- Display the assembly-language instructions of the program

- Stop program execution at any point
- Examine the values of program variables, given their addresses
- Modify the value of any program variable
- Execute a program one instruction at a time
- Display the values of machine registers
- Modify the values of machine registers

The adb debugger can be used to debug programs at all optimization levels, including vector code and programs running on multiple processors.

For a detailed description of adb and complete instructions on its use on C Series machines, refer to the *CONVEX adb (Assembly-Language Debugger) User's Guide.*

Visual debugger (CXdb)

CONVEX CXdb is an optional window-based debugger that includes all the functionality of regular debuggers and is capable of debugging parallel and optimized code. CXdb runs on both CONVEX C Series and CONVEX SPP Series computers.

CXdb can debug any CONVEX Fortran or C executable; however, to fully employ the power of CXdb, the program should be compiled with the -cxdb command line option. Failure to compile with -cxdb will prevent access to symbolic debugging information.

CXdb can perform these functions, among others:

- Source-level debugging of parallel and optimized code
- Display debugging contexts for source code and disassembled code in a windowing environment, providing the following windows:
  - Process interface window
  - Command window
  - Source Code window
  - Help window
  - Stack Trace window (X Window mode only)
  - Memory Display window (X Window mode only)
  - Register windows
  - Assembly Code window
  - Thread Activity window

- Access program variables by name
- Examine core files
- Set breakpoints, tracepoints and conditional eventpoints
- Debug program source code or disassembled code
- Examine and modify program variables, registers and the stack
- Step process execution line-by-line
- Debug programs containing multiple source modules
- Debug mixed-language programs
- Modify the environment in which your process runs

CXdb's windowing environment supports both line-oriented terminals and X Window–capable workstations. This windowing environment eases the task of debugging programs that contain multiple threads of execution.

Refer to Chapter 1, "Compiling programs," for additional information on using the -cxdb command-line option with the CONVEX Fortran compiler.

For more detailed information about the CXdb product, consult *CXdb Commands and Parameters* or *CXdb Concepts, Windows, and Messages*.

Performance analyzer (CXpa)

The performance analyzer, CXpa, is an optional product that gathers and analyzes profiling data. The performance analyzer is an interactive tool that can display four types of reports about a program's activity:

- **Basic Block Report** — Displays basic block performance data, including the number of times a block was executed, the total number of blocks in a routine, and other information.
- **Loop Report** — Displays loop performance data, including the iteration count for each loop, the CPU time for each loop, optimizations performed on each loop, and other statistics.
- **Parallel Region Report** — Displays parallel region performance data, such as the CPU time of each region, the number of times each region was executed, and other data.
- **Routine Report** — Displays routine performance data, including the CPU time of each routine, Mflops, and other data.

CXpa provides a graphic (X Window) interface, a line mode (VT100) interface, and a batch mode that enables you to run CXpa from a shell script.

To use the performance analyzer, you first must compile your program with either the `-cxpa` or the `-cxpab` compiler option. The `-cxpa` option instruments the compiled code so that its performance can be measured at the routine level and loop level; the `-cxpab` option instruments the compiled code so that its performance can be measured at the block level. The instrumentation produced by the `-cxpa` and `-cxpab` options is incompatible. Specifying both `-cxpa` and `-cxpab` for the same file is not allowed.

The `-cxpamon` compiler option allows users to select a specific monitor instrumentation library by specifying the directory where the library resides. This compiler option is provided to facilitate using multiple versions of CXpa on a system.

The `-cxpalib` compiler option instructs the compiler to link your program with libraries instrumented for use with CXpa. Programs profiled with `-cxpalib` may execute much slower, and any errors in the instrumentation of the libraries may cause the program to core dump. The amount of profiling data under this option is significantly increased.

The performance analyzer is described in detail in *CXpa Reference*.

Profilers

Three profilers are available to monitor the performance of your program:

- Standard profiler (`prof`)
- Basic block profiler (`bprof`), available only on C Series machines
- Graph profiler (`gprof`)

You can use the information obtained from a profiler to improve the efficiency and speed of a program. To use a specific profiler, you must first compile your program using one of the options described in Table 10.

Table 10 Compiler options for profiling

Compiler option	Description
-p	Produces code that counts the number of times each utility is called. When the program begins execution, the monitor utility is called. If the program completes normally, a profile file (mon.out) is produced. This file can then be processed by the prof profiler to generate an execution profile. When you specify the -p option, the loader uses profiling libraries instead of the standard libraries.
-pb	This option is available only on C Series machines. Produces code that counts the number of times each statement is executed. If the program completes normally, a basic block profile file (bmon.out) is produced. This file can then be processed by the bprof profiler to display the source-level execution counts.
-pg	Produces counting code in the manner of -p, but invokes a runtime recording mechanism that keeps more extensive statistics. If the program completes normally, a call graph profile file (gmon.out) is created. This file can then be processed by the gprof profiler to produce a comprehensive execution profile.

CXtrace (SPP Series only)

CXtrace is a set of software tools for gathering and displaying information about the performance of your programs. CXtrace has three main components: a source-code instrumentor, a runtime performance monitoring library, and a set of analysis tools.

The CXtrace source-code instrumentor (xinstrument) inserts performance monitoring routines into an application's source code, allowing the programmer to select the files and constructs that are instrumented.

A set of monitoring routines comprises CXtrace's runtime performance monitoring library. These routines measure and

record aspects of a program's performance, such as message-passing overhead, synchronization overhead, and time spent in different subroutines.

CXtrace's analysis tools include View Kernel (VK), which animates an application's behavior and allows you to observe implementation bottlenecks and load imbalances; trace view (`tv`), which provides a static view of an entire trace file; and `tally`, which provides performance statistics for an entire program.

For more information about the CXtrace product, contact your local CONVEX sales representative.

Consultant III and CXtools

Two packages of CONVEX software tools are available for C Series and SPP Series programmers. These packages, Consultant III and CXtools, provide tools that facilitate debugging and analyzing the performance of Fortran and C programs.

Consultant III, which is available only for use with CONVEX C Series machines, includes a suite of profilers (`prof`, `bprof`, and `gprof`), CXdb, and CXpa.

CXtools, which is available for CONVEX SPP Series computers, includes CXdb, CXpa, and CXtrace.

Each of the programs included in these packages is summarized in sections of this chapter. For more information about either Consultant III or CXtools, please contact your local CONVEX sales representative.

Application Compiler

The CONVEX Application Compiler is an interprocedural analyzer that tracks the flow of data and control between procedures. The information generated by this analysis removes scope restrictions on optimization, which allows the Application Compiler to generate more efficient code by taking the entire program, with all its dependencies, into account. The database of program information that the interprocedural analyzer builds also allows the Application Compiler to perform better error checking, leading to more robust and reliable programs.

On C Series machines, many of the optimizations discussed in the *CONVEX Fortran Optimization Guide* are performed automatically by the Application Compiler, with little or no user intervention.

The CONVEX Application Compiler is an optional product. For more information, refer to the *CONVEX Application Compiler User's Guide*, or contact your CONVEX sales representative.

Postmortem dump (C Series only)

The postmortem dump (pmd) produces a core dump if the program running under it aborts. A core dump is a copy of the executable image in memory at the point when it aborted. To run a program under pmd, you must first compile the program using the -db option.

At your option, pmd produces either a short- or long-form dump containing the information shown in Table 11. If you do not indicate a choice, the short-form dump is the default.

Table 11 Postmortem dump contents

Type of dump	Contents
Short form	The signal that caused the program to abort; a runtime stack backtrace; the approximate source line location at which the exception occurred.
Long form	The signal that caused the program to abort; a runtime stack backtrace; the approximate source line location at which the exception occurred; the contents of the machine registers; a dump of active local variables in each routine on the runtime stack; a dump of global, or common, variables; the region of disassembled object code where the exception took place; a summary of resources used by the program.

The summary of resources produced by the long-form dump includes execution time, elapsed time, percent of time in CPU, size of shared memory and unshared memory, page faults, and swaps.

`error` utility

The `error` utility takes the error messages produced by the compiler and inserts them into the source at the point at which the error occurred.

The `error` utility is run by piping the diagnostic output generated by `fc` to it. To do this under `csh`, append the characters `|& error` to the end of the compile command.

```
fc f.f |& error
```

pipes the standard error messages through the `error` utility. Because `error` alters your source file, you may want to make a copy of the source file before you use it.

Note

The `-LST` command line option works similarly to the `error` utility and is easier to use. Refer to the "Messages and listing options" section in Chapter 1 of this guide for more information on `-LST`.

Development tools

Input/output operations

3

This chapter presents an overview of the key input/output (I/O) concepts and features of CONVEX Fortran. The information in this chapter is reiterated and elaborated upon in *Fortran Language Reference* Chapter 9.

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 (HP) I/O than does C Series Fortran. For HP compatibility information, see *Fortran Language Reference* 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 14 and Table 15 on page 72 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 12 lists the I/O statements CONVEX Fortran supports by category.

Table 12 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 12 (continued) Input/output statements

Type	Statement	Use
Output (cont.)	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. This statement is available only on C Series machines.

Supported I/O methods

Table 13 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 13 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 13 (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, `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 `FOR`*nnn*`OPEN`, 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.

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.

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 16 on page 73 shows the general forms of CONVEX Fortran I/O statements in which the unit is specified implicitly.

C Series only

Table 14 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.

Table 14 C Series implicit unit numbers (C Series only)

File	Default unit number
stdin	5
stdout	6
stderr	0

SPP Series only

Table 15 lists the default SPP Series unit numbers CONVEX Fortran assigns to stdin for input, stdout for output, and stderr for error messages.

Table 15 SPP Series implicit 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 16 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 16 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

I/O operations

Calling conventions

4

This chapter describes the methods CONVEX Fortran uses to call subprograms. It also explains how to call routines written in languages other than Fortran.

Subprogram calls in CONVEX Fortran use the same calling conventions as those used by other CONVEX language processors. These conventions permits Fortran programs to call routines written in CONVEX assembly language, C, or Ada, and vice versa.

Fortran subprogram calling convention

The basis of the calling standard is the passing of actual arguments or parameters. The standard supports two argument-passing mechanisms: call-by-value and call-by-reference.

Although Fortran arguments are passed by reference, CONVEX Fortran provides built-in functions (`%VAL` and `%REF`) to support both mechanisms. By default on C Series machines, and when `-nore` compiler option is specified on SPP Series machines, a routine call involves passing a pointer to the list of arguments, called the *argument packet*.

Fortran argument packets

On CONVEX C Series machines, the default method of passing arguments to subroutines is to use argument packets.

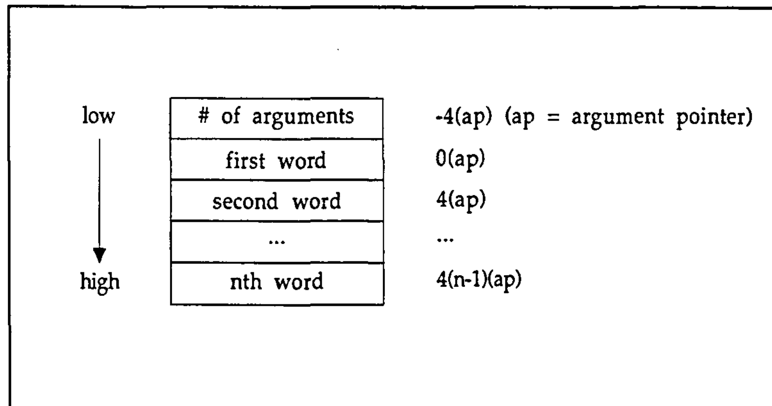
SPP Series only

On SPP Series machines, however, arguments are passed on the stack by default. To pass arguments using the argument packet method on SPP Series machines, you must specify the `-nore` option. For more information about default argument passing methods, see the `-re` and the `-nore` compiler option descriptions in Chapter 1.

An argument packet is a sequence of word (4-byte) entries. Only precompiled argument lists are allocated static memory space.

When possible, the compiler creates argument list entries at compile time, but it cannot precompile an argument list if any argument is a dummy argument or array element with nonconstant subscripts. Figure 11 shows the layout of an argument packet in memory.

Figure 11 Argument packet: example 1



The first word is normally the address of the first argument, the second word is the address of the second argument, and so on. For character arguments, an extra by-value word containing the length of the character entity is added to the end of the list. For each character argument there is one extra word which occurs in the same order as the character argument addresses.

C Series only

On C Series machines, for functions that return character and complex values, an extra argument is added before the first user-specified argument to receive the function result. For a character-valued function, this extra argument contains two words: the first is the address of the character string to receive the value of the function and the second is its maximum length.

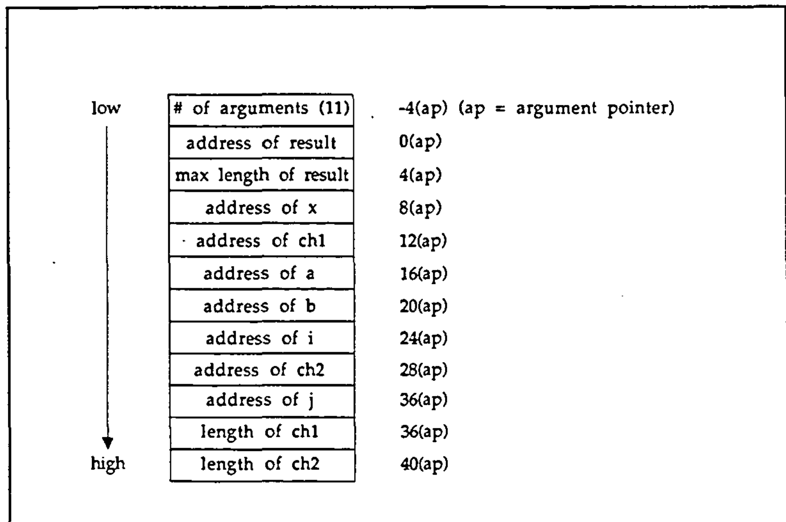
SPP Series only

On SPP Series machines, character string arguments are passed such that a list of all character arguments is followed by a list that contains the length of each character argument. This argument length list is appended to the end of the parameter list.

Figure 12 shows the argument packet layout that C Series Fortran generates for the following Fortran code:

```
CHARACTER*(*) FUNCTION F (X,CH1,A,B,I,CH2,J)
CHARACTER*10 CH1
CHARACTER*5  CH2
REAL A
REAL*8 B
COMPLEX X
INTEGER*4 J
INTEGER*2 I
END
```

Figure 12 Argument packet: example 2 (C Series only)



Argument-passing mechanisms

The CONVEX calling standard supports two methods of passing arguments to subprograms:

- Passing arguments by reference, where the argument-packet entry is the address of the value. This is the default method in Fortran. Passing by reference also is supported through the `%REF` function.
- Passing arguments by immediate value, where the argument-packet entry is the value. This method is supported by the `%VAL` function.

Argument packet built-in functions

It is not always possible to use the default Fortran calling convention to pass arguments to routines not written in Fortran. CONVEX Fortran provides the built-in functions `%VAL` and `%REF` for use in these cases. These functions can only appear in actual argument lists.

Table 17 Built-in functions and defaults for argument lists

Data type expressions	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

%REF function

This built-in function ensures that the argument packet entry uses the reference mechanism. It is represented as:

```
%REF(arg)
```

The argument packet entry is the address of the actual argument, *arg*. The argument value can be a numeric or character expression, array, or procedure name. Refer to Table 17 for details concerning values and the size of the argument.

%VAL function

This built-in function ensures that the argument packet entry uses the immediate value mechanism. It is represented as:

```
%VAL(arg)
```

The argument packet entry is the value of the actual argument, *arg*. Refer to Table 17 for details concerning values and the size of the argument.

```
CALL SUB(3,%VAL(10))
```

In this example, the first constant is passed by reference, the second by immediate value.

Function return values

The method of returning a value of a function depends on the data type of the value, as the following tables summarize. C Series Fortran and SPP Series Fortran use different methods. Table 18 lists C Series methods and Table 19 lists the methods used on SPP Series.

Table 18 C Series function return values (C Series only)

Data type	Return method
LOGICAL INTEGER REAL	Scalar register S0
REAL*16	If the function returns a REAL*16, the first word of the argument list is the address of the result.
COMPLEX	If the function returns a complex, the first word of the argument list is the address of the result.
CHARACTER	If the function returns a character, the first word of the argument list is the address of the result and the second word is the length of the result.

Calling conventions

CONVEX SPP Series Fortran supports Hewlett-Packard's PA-RISC procedure calling conventions.

Table 19 SPP Series function return values (SPP Series only)

Data type	Return method
LOGICAL INTEGER COMPLEX	General register gr28 and, for 33 to 64 bit function results (INTEGER*8, LOGICAL*8, COMPLEX*8), register gr29. COMPLEX*16 function results are returned by placing the result's address in gr28.
REAL	Floating-point register fr4 and, for REAL*8 values, register fr5.
REAL*16	If the function returns a REAL*16, the first word of the argument list is the address of the result. (Same as C Series.)
CHARACTER	The function returns a list of all character strings, followed by a list containing the length of each character string.

%LOC function

The %LOC function returns the address of a storage element as an INTEGER*4 value. %LOC can only be used in an arithmetic expression. It is represented as:

```
%LOC(arg)
```

where *arg* is a storage element.

```
I = %LOC(J)-4
```

In the example, I now holds in storage the address of the word preceding J. %LOC is useful when passing argument data structures containing the address of storage elements to non-Fortran routines.

Non-Fortran-to-Fortran calling sequence

If you are writing in assembly language or C and want to call a Fortran routine, follow the procedures outlined below. Figure 13 is an example of code you might write in assembly-language or C to call a Fortran subroutine.

Perform the following sequence of steps to call a Fortran routine from assembly language or C:

1. Push the arguments onto the runtime stack in reverse order.
2. Update the argument pointer to point to the top of the runtime stack.
3. On C Series machines only, push an additional argument, the number of arguments, onto the stack.
4. Call the subroutine (using the `calls` instruction).

Where possible, the arguments are precompiled and the calling sequence is reduced to the following two steps:

1. Load the address of the argument packet into the argument pointer.
2. Call the subroutine (using the `calls` instruction).

Note that the assembly-language instructions presented in Figure 13 were generated on a CONVEX C Series machine; assembly code for SPP Series machines will differ.

Figure 13 Calling a Fortran subroutine (C Series only)

```
CALL SUB(A,B,C)   !Fortran call with three real arguments
                  ! NOTE: This is C Series code--SPP Series code differs
                            ! equivalent assembler calling sequence:
pshea   c                   ! push the third argument's address
pshea   b                   ! push the second argument's address
pshea   a                   ! push the first argument's address
mov     sp,ap               ! set up the argument pointer
pshea   3                   ! push number of words in argument list
calls   _sub_               ! call the subroutine
ld.w    12(fp),ap           ! restore the argument pointer
add.w   #16,sp              ! restore the stack pointer

sub_ (&a,&b,&c);              /* equivalent C call */
```

C Series only

On C Series machines, the arguments are pushed in reverse, because the stack grows toward low memory addresses. Thus, the last argument pushed ends up at the top of the list.

SPP Series only

On SPP Series, the stack grows toward higher memory addresses. Arguments appear in order first to last, from higher memory addresses to lower addresses.

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 20, where *name* represents the symbolic name.

Table 20 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 20 shows, different external names are generated for code compiled on 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. The "Interlanguage programming examples (C Series only)" section of this chapter contains several sample programs that illustrate how to reference external names from both Fortran and C programs.

Table 20 lists the external names generated by the C compiler on both CONVEX C Series and CONVEX SPP Series computers.

Table 21 CONVEX C external naming conventions

C program block	C Series external name	SPP Series external name
Main program	_main	main
Subprogram	_*name*	*name*

Note that if your main program is not in Fortran, then Fortran units 5, 6, and `stderr` (either unit 0 on C Series machines, or unit 7 on SPP Series) are not connected and data format conversions will not work. Signal handling depends on the language used in the main program.

C Series external naming

On CONVEX C Series machines, the C language prepends a single underscore to function names and external variable names (see Table 20). For example, on a C Series machine a reference to a Fortran `COMMON` block symbolically named `FFT` from C is written `_fft_` for the resulting name to match `__fft_`, the external name generated by Fortran.

SPP Series external naming

On SPP Series machines, the C language does not add underscores to function names or external variable names (see Table 20). For instance, a reference from C to a Fortran subroutine symbolically named `CANSAT` must be written `cansat` (with no leading or trailing underscores) for the resulting name to match `cansat`, the external name generated by the Fortran compiler.

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

Calling conventions

which user-written subprograms are named. This option is available only on SPP Series machines.

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.

Data representations

Table 22 shows corresponding Fortran and C declarations. In Fortran, all variables declared as `INTEGER`, `LOGICAL`, or `REAL` without an explicitly declared size occupy the same amount of memory.

The following table does not list the CONVEX SPP Series data types `GATE` and `BARRIER` because they normally are manipulated only by intrinsic procedures.

Table 22 Fortran and C declarations

Fortran	C
`INTEGER*1 x`	`char x;`
`INTEGER*2 x`	`short int x;`
`INTEGER x`	`int x;`
`INTEGER*8 x`	`long long int x;`
`LOGICAL*1 x`	`char x;`
`LOGICAL*2 x`	`short int x;`
`LOGICAL x`	`long int x;`
`LOGICAL*8 x`	`long long int x;`
`REAL x`	`float x;`
`REAL*8 x`	`double x;`
`DOUBLE PRECISION x`	`double x;`

Table 22 (continued) Fortran and C declarations

Fortran	C
`REAL*16 x`	No corresponding C type.
`COMPLEX x`	`struct {float r, i;} x;`
`COMPLEX*16 x`	`struct {double dr, di;} x;`
`DOUBLE COMPLEX x`	`struct {double dr, di;} x;`
`CHARACTER*6 x`	`char x[6];`

Return values

A function of type INTEGER, LOGICAL, REAL, or DOUBLE PRECISION declared as a C function returns the corresponding type.

SPP Series only

On SPP Series machines, a COMPLEX (COMPLEX*8) function returns values in the general-purpose return registers gr28 and gr29. DOUBLE COMPLEX (COMPLEX*16) return values are returned via the hidden argument, which is passed in gr28.

C Series only

On C Series machines, a COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine with an additional argument pointing to where the return value is to be stored. Refer to Table 23.

Table 23 Complex function: C equivalent (C Series only)

Complex Fortran function	Equivalent C function

`COMPLEX FUNCTION F(...)`
.
.
.
	`void f_(temp,...)`
	`struct {float r, i;} *temp;`
	.
	.
	.

C Series only

On C Series machines, a character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Refer to Table 24.

Table 24 Character functions: C equivalent (C Series only)

Fortran character function	Equivalent C function	C language invocation

`CHARACTER*15 FUNCTION G(...)`	`void g_(result,length,...)`
	`char result[];`
	`long int length;`
	.
	.
	.
		`char chars[15]`
		.
		.
		.
		`g_(chars,15L,...);`
		.
		.
		.

Subroutines are invoked as if they were integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
CALL NRET(*1, *2, *3)
```

is treated as if it were the computed GOTO statement

```
GOTO (1, 2, 3), NRET( )
```

Argument packets (C Series only)

All Fortran arguments are passed by address. For every argument that is of type CHARACTER or that is a dummy procedure of type CHARACTER, an argument giving the length of the value is passed (string lengths are integer quantities passed by value). The order of arguments is:

1. Extra arguments for complex and character functions
2. Address for each datum or function
3. A `long int` for each character argument

An example is shown in Table 25.

Table 25 Character arguments: C equivalent (C Series only)

Fortran call	Equivalent C call
EXTERNAL I	int i( );
CHARACTER*7 S	char s[7];
INTEGER B(3)	long int b[3];
.	.
.	.
.	.
CALL SAM(I, B(2), S)	sam_(i, &b[1], s, 7L);

The first element of a C array always has subscript 0, while Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C arrays are stored in row-major order.

Examples (C Series only)

The examples in this section illustrate how to interface to Fortran from other languages and how to use argument-passing techniques.

Equivalent Fortran and non-Fortran code (C Series only)

The following examples were created on CONVEX C Series machines; code written on CONVEX SPP Series machines will differ from the code in these examples. The addresses represented in assembly-language code segments can vary from compilation to compilation.

Example 1 — simple procedure

This example shows a simple Fortran procedure and how it is written in C and assembly language if called from a Fortran program.

Fortran source code:

```
SUBROUTINE SUB (I,R,D)
INTEGER I
REAL R
DOUBLE PRECISION D
D = I + R
END
```

Calling conventions

C source code (C Series):

```
sub_(i,r,d)
int *i;
float *r;
double *d;
   {
   *d = *i + *r;
   }
```

Note that the procedure's symbolic name (sub_) has a trailing underscore, which is added so the resulting external name (_sub_) is equivalent to the Fortran procedure's external name on C Series machines.

Assembly-language code (C Series):

```
        .
        .
        .
sub:ld.w    @0(ap),s0 ; s0 = i
    ld.w    @4(ap),s1 ; s1 = r
    cvtw.s  s0,s0     ; convert i to real
    add.s   s1,s0     ; add it to r
    cvts.d  s0,s0     ; convert result to
                      ; real*8
    st.l   s0,@8(ap) ; store the result in d
```

Example 2 — two character arguments

This example shows a call involving two character arguments separated by other arguments and the corresponding compiler-generated assembly code.

Fortran source code:

```
SUBROUTINE SUB1
CHARACTER*5 A,B
REAL X,Y
CALL CHARARGS (A,X,Y,B)
END
```

Assembly-language code (C Series):

```
        .
        .
        .
LC:  ds.w    6              ; 6 arguments
     ds.w    LU             ; address of A
     ds.w    LU+12          ; address of X
     ds.w    LU+16          ; address of Y
     ds.w    LU+5           ; address of B
     ds.w    5              ; length of A
     ds.w    5              ; length of B

ldea         LC+4,ap        ; load packet pointer
calls        _charargs_
```

Example 3 — character function

This example shows a call to a function that returns a character value and the corresponding compiler-generated assembly-language code.

Fortran source code:

```
SUBROUTINE SUB2
CHARACTER*10 A,F
A = F(1.7)
END
```

Assembly-language code (C Series):

```
        .
        .
        .
sub.w       #16,sp         ; set up local frame
pshea       LC             ; 1.7
pshea       10             ; length of result
pshea       -12(fp)        ; address of result
; can vary from compilation to compilation
mov         sp,ap          ; initialize ap
pshea       3              ; # of return arguments
calls       _f_            ; call function F
add.w       #16,sp         ; reset sp
ldea        LU,a5          ; variable A
ldea        -12(fp),a1     ; result address
ld.1        0(a1),s0       ; load result
```

```
st.l        s0,0(a5)    ; store result
ld.h        8(a1),s0    ; load result
st.h        s0,8(a5)    ; store result
st.h        s1,0x8(a5)
```

Example 4 — COMPLEX function

This example shows a call to a function that returns a COMPLEX value and the corresponding compiler-generated assembly code.

Fortran source code:

```
SUBROUTINE SUB3
COMPLEX X,F
X = F(10)
END
```

Assembly-language code (C Series):

```
   .
   .
   .
pshea    LC+32       ; address of argument 10
pshea    -8(fp)      ; address of function result
mov      sp,ap       ; packet address
pshea    2           ; number of arguments
calls    _f_         ; call function f
ld.w     12(fp),ap ; restore ap
add.w    #12,sp      ; restore sp
```

Example 5 — argument passing

This example illustrates how subprogram arguments are passed.

Fortran source code:

```
SUBROUTINE SUB4
EXTERNAL F
CALL USEIT (F,X)
END
```

Assembly-language code (C Series):

```
        .
        .
        .
LC:   ds.w    2       ; 2 arguments
      ds.w    _f_     ; addr. of user ext. function f
      ds.w    LU+40 ; addr. of X
        .
        .
        .
      ldea    LC+4,ap
      calls   _useit_
```

Example 6 — array arguments

This example shows how an array argument is passed.

Fortran source code:

```
SUBROUTINE SUB5
REAL A(20)
CALL USEARRAY (A,X,Y)
END
```

C Series assembly-language code:

```
        .
        .
        .
LC:   ds.w     3       ; 3 arguments
      ds.w     LU+44 ; address of A
      ds.w     LU+124; address of X
      ds.w     LU+128; address of Y
        .
        .
        .
      ldea     LC+4,ap
      calls    _usearray_
      ld.w     12(fp),ap
```

Calling conventions

Interlanguage programming examples (C Series only)

The programs in this section illustrate how to interface your Fortran code with code written in the C programming language.

The examples in this section were created on CONVEX C Series machines; code written on CONVEX SPP Series machines will differ from the code in these examples.

Example 1 — passing INTEGER and REAL data

This example shows how to pass INTEGER*4 and REAL*4 Fortran data types to a C function.

Fortran source code:

```
PROGRAM MAIN
! Source file is main_ir.f
INTEGER*4 I
REAL*4 A
CALL SUB1(I,A)
PRINT *, I, A
END
```

C source code (C Series):

```
void sub1_ ( int *j, float *b)
/* Source file is sub1_ir.c */
{
 *j = 90125;
 *b = 521.125;
}
```

Parameters b and j in the C source code are declared as pointers because Fortran passes parameters by reference. The C procedure's symbolic name (sub_) has a trailing underscore because the C compiler prepends an underscore when creating the external name. The resulting external name (_sub_) is equivalent to the Fortran procedure's external name on C Series machines.

The command lines to compile and run this code are:

```
% cc -c sub1_ir.c
% fc -sa main_ir.f sub1_ir.o
% a.out
     90125   521.1250
```

Example 2 — passing COMPLEX data

This example demonstrates how to pass a COMPLEX data type from a Fortran main program to a C function.

Fortran source code:

```
PROGRAM MAIN
! Source file is main_cx.f
COMPLEX*8 CX
CALL SUB1(CX)
PRINT *, CX
END
```

C source code (C Series):

```
void sub1_ (struct { float r, i;} *cxs )
/* Source file is sub1_cx.c */
{
 cxs->r = 32.45;
 cxs->i = 76.89;
}
```

The equivalent C data type for the COMPLEX data type is a `struct`.

The C procedure's symbolic name (`sub1_`) has a trailing underscore so the resulting external name (`_sub1_`) matches the Fortran procedure's external name on C Series machines.

The command lines to compile and run this code are:

```
% cc -c sub1_cx.c
% fc -sa main_cx.f sub1_cx.o
% a.out
     ( 32.45000     , 76.89000 )
```

Example 3 — passing CHARACTER data

This example illustrates how to pass the CHARACTER*n data type from a Fortran main program to a C function.

Fortran source code:

```
PROGRAM MAIN
! Source file is main_str.f
CHARACTER*11 CVAR1
CHARACTER*6 CVAR2
CALL SUB1(CVAR1, CVAR2)
PRINT *, CVAR1, CVAR2
END
```

C source code (C Series):

```
#include <string.h>
void sub1_ (char *str1, char *str2, int d1, int d2)
/* Source file is sub1_str.c */
{
 (void) strncpy (str1, "12345567890", d1);
 (void) strncpy (str2, "abcde", d2);
}
```

The CHARACTER variables in the Fortran code are declared 1 character longer than the strings that are copied into them in the C source file. This is done because C strings include a null terminator character. You can corrupt memory if you attempt to write a string into a space that is not large enough.

The C function parameters d1 and d2 correspond to the declared lengths of the Fortran variables CVAR1 and CVAR2; they are not pointers because the Fortran compiler places their *values* on the call stack.

The C procedure's symbolic name (sub_) has a trailing underscore because the C compiler prepends an underscore when creating the external name. The resulting external name (_sub_) is equivalent to the Fortran procedure's external name on C Series machines.

The command lines to compile this example and its output are:

```
% cc -c sub1_str.c
% fc -sa main_str.f sub1_str.o
% a.out
    12345567890abcde
```

Example 4 — accessing COMMON blocks

The following example illustrates how to access Fortran COMMON blocks from C.

Fortran source code:

```
PROGRAM MAIN
! Source file is main_com.f
REAL*8 A, X
INTEGER*4 B, Y
COMMON A,B
COMMON /NAMED/ X, Y
CALL SUB1()
PRINT *, A, B
PRINT *, X, Y
END
```

C source code (C Series):

```
struct block {double a; int b;};
/* Source file is sub1_com.c */
void sub1_ ()
{
 extern struct block __blnk_;
 extern struct block _named_;

 __blnk_.a = 3.141592654;
 __blnk_.b = 46616;

 _named_.a = 56.6568934;
 _named_.b = 235;
}
```

In the C code, structures are used to access COMMON block areas defined in the Fortran code.

C references to the blank (unnamed) COMMON block areas are __blnk_ because the C compiler prepends an underscore when creating the external name. The resulting external name (___blnk_) matches the corresponding external name generated by the Fortran compiler.

C refers to the other COMMON block as _named_; the compiler-generated external name for it (__named_) matches the external name Fortran generates for NAMED.

Calling conventions

The command lines to compile and run this code are:

```
% cc -c subl_com.c
% fc main_com.f subl_com.o
% a.out
     3.14159265400000         46616
     56.6568934000000         235
```

Exemple 5 — REAL and INTEGER functions

This example demonstrates how to make calls from C to Fortran functions that return `int` and `float` data types.

Fortran source code:

```
INTEGER FUNCTION IADDEM (J1, J2)
! Source file for both functions is funs_infl.f
INTEGER J1, J2
IADDEM = J1 + J2
RETURN
END

REAL FUNCTION ADDEM (B1, B2)
REAL B1, B2
ADDEM = B1 + B2
RETURN
END
```

C source code (C Series):

```
#include <stdio.h>
extern float addem_ (float *a1, float *a2);
extern int iaddem_ (int *i1, int *i2);

/* Source file is main_infl.c */

main ( )
{
 int i1 = 43;
 int i2 = 12;
 float a1 = 43.;
 float a2 = 12.;
 (void) printf("%d\n", iaddem_ (&i1, &i2) );
 (void) printf("%f\n", addem_ (&a1, &a2) );
}
```

The C data types compatible with the default Fortran REAL and INTEGER data types are `float` and `int`, respectively. The C

external function declarations specify these data types, which match the data types returned by the Fortran functions.

The functions declared in the C source code are referenced with a trailing underscore (addem_ and iaddem_) so that the external name C generates will match the external names Fortran generates (_addem_ and _iaddem_).

The command lines to compile and run this code are:

```
% fc -c funs_infl.f
% cc main_infl.c funs_infl.o
% a.out
55
55.000000
```

Example 6 — string functions

This example demonstrates how to call a Fortran function that returns a string from a C main program.

Fortran source code:

```
CHARACTER*20 FUNCTION MAKESTR (CVAR1)
! Source file is fun_str.f
CHARACTER*5 CVAR1
MAKESTR = CVAR1//' edcba'//CHAR(0)
RETURN
END
```

C source code (C Series):

```
#include <stdio.h>
/* Source file is main_str.c */
extern void makestr_ (char cstring[],
                long int cstr_len, char *arg1,
                long int arg1_len);
main ()
{
 char cstring[20];
 char wxyz[] = " wxyz";
 makestr_( cstring,20L, wxyz, sizeof(wxyz)-1);
 (void) printf("%s\n", cstring);
}
```

Although a Fortran function is called in the C source code, its return type in the C declaration of makestr_ is type void.

Calling conventions

The Fortran function, MAKESTR, concatenates two strings and appends the null character (\0) required by C using the CHAR() function.

The first parameter of makestr_ is a pointer to the character variable returned by the Fortran function. The second parameter (cstr_len) contains the amount of space allocated by the C main program for the character variable returned by the Fortran function. The third parameter (arg1) is a pointer to a string that MAKESTR uses.

The last parameter of makestr_ (arg1_len) contains the length of arg1. In this example, it is computed using the sizeof macro, which returns the number of bytes in an array operand. One (1) is subtracted from the length to strip the null character (\0) from the end of the string.

The command lines to compile and run this program are:

```
% fc -c fun_str.f
% cc main_str.c fun_str.o -lF77
% a.out
 wxyz edcba
```

In the Fortran source code, the string concatenation in the assignment to MAKESTR uses an intrinsic function for$s_cat. Because this function is contained in the Fortran library libF77.a, you must include the -lF77 compiler option at the end of the cc command line to compile this example.

System utilities

5

This chapter describes CONVEX Fortran's system utility routines, which provide a runtime interface between CONVEX Fortran programs and the operating system. On C Series machines, these routines interface between Fortran and the ConvexOS operating system. On SPP Series computers, the interface is between Fortran and the SPP-UX operating system.

The library in which the utility routines reside (`libU77.a`) includes useful character and math functions. On CONVEX C Series machines this library is `/usr/lib/libU77.a`. On SPP Series machines it is `/usr/convex/fc`*Ver*`/lib/libU77.a` (where *Ver* is the version number of the compiler). Any referenced utility is automatically loaded during linking.

How to call utility routines

A utility routine is called in the same manner as a user-written subroutine. An example is the `chdir` function listed below and fully described in the chdir(3F) man page. (The man page gives the name, synopsis, description, and file location of each utility.) The synopsis gives the information required for referencing the utility:

```
INTEGER FUNCTION CHDIR (dirname)
CHARACTER*(*) dirname
```

`CHDIR` is a function that returns an integer value, and you must pass it one parameter (a directory name) in a character variable of arbitrary length. The following program uses `CHDIR` to change to the `/tmp` directory:

```
INTEGER*4 FUNCTION CHDIR
CHDIR ("/tmp")
END
```

Note

The routines described in this chapter that accept `INTEGER*4` arguments must always be passed `INTEGER*4` arguments, even if the `-p8`, or `-pd8` option is set. Similarly, routines that accept `REAL*4` arguments must be passed `REAL*4` arguments regardless of which `-p` option is set.

Operating system utilities

The calling sequences for the routines shown in Table 26 are also described in section 3F of the man pages under the heading indicated in the table.

Routines marked with an asterisk (*) are available only on C Series machines.

Table 26 Calling sequences for ConvexOS and SPP-UX utilities

Fortran routine	Man page (3F)	Description
abort	abort	Terminate with memory image
access	access	Determine accessibility of file
alarm	alarm	Execute a subroutine after a specified time
bessel	bessel	Calculate Bessel functions of two kinds for integer orders
chdir	chdir	Change default directory
chmod	chmod	Change mode of file
ctime	stime	Return system time
dffrac	flmin	Return fractional accuracy of double-precision float
dflmax	flmin	Return maximum positive double-precision float
dflmin	flmin	Return minimum positive double-precision float
drand	rand	Return random values
dtime	etime	Return elapsed execution time since last call to dtime
errtrap*	errtrap	Enable or disable certain signal traps
etime	etime	Return elapsed execution time
exit	exit	End process with status
fdate	fdate	Return date and time in ASCII string
ffrac	flmin	Return fractional accuracy of single-precision float
fgetc	getc	Get a character from a logical unit

Table 26 **(continued)** Calling sequences for ConvexOS and SPP-UX utilities

Fortran routine	Man page (3F)	Description
flmax	flmin	Return maximum positive single-precision float
flmin	flmin	Return minimum positive single-precision float
flush	flush	Flush output to a logical unit
fork	fork	Create a copy of this process
fputc	putc	Write a character to a Fortran logical unit
fseek	fseek	Reposition file on logical unit
fstat	stat	Get file status
ftell	fseek	Reposition file on logical unit
gerror	perror	Get system error message
getarg	getarg	Return command line arguments
getc	getc	Get a character from a logical unit
getcwd	getcwd	Get current working directory
getenv	getenv	Get value of environment variables
getgid	getuid	Get group ID of caller
getlog	getlog	Get user login name
getpid	getpid	Get process id
getuid	getuid	Get user ID of the caller
gmtime	stime	Return system time
hostnm	hostnm	Return name of current host
iargc	getarg	Return command line arguments
ierrno	perror	Get system error messages
inmax	flmin	Return the maximum positive integer value
ioinit	ioinit	Change default settings of I/O attributes
irand	rand	Return random values
isatty	ttynam	Find name of terminal port
itime	idate	Return date or time in numerical form
kill	kill	Send a signal to a process

Table 26 **(continued)** Calling sequences for ConvexOS and SPP-UX utilities

Fortran routine	Man page (3F)	Description
link	link	Make a link to an existing file
lnblnk	rindex	Return index of last nonblank character in a string
loc	loc	Return the address of an object
longjmp	longjmp	Restore stack environment
lstat	stat	Get file status
ltime	stime	Return system time
perror	perror	Get system error messages
putc	putc	Write a character to a Fortran logical unit
qsort	qsort	Perform quick sort
qffrac	flmin	Return fractional accuracy of quad-precision float
qflmax	flmin	Return maximum positive quad-precision float
qflmin	flmin	Return minimum positive quad-precision float
qrand	rand	Return random values
rename	rename	Rename a file
rindex	index	Return index of last occurrence of a substring
setjmp	setjmp	Save stack environment
signal	signal	Change the action for a signal
sleep	sleep	Suspend execution for an interval
stat	stat	Get file status
stime	stime	Return system time
system	system	Execute an operating system command
symlnk	link	Make a symbolic link to an existing file
time	time	Return time in an ASCII string
topen	topen	Provide low-level interface for magnetic tape devices
traceback*	traceback	Print names of routines in call stack
traper*	traper	Trap floating-point underflow and integer overflow
ttynam	ttynam	Find name of a terminal port

Table 26 (continued) Calling sequences for ConvexOS and SPP-UX utilities

Fortran routine	Man page (3F)	Description
unlink	unlink	Remove a directory entry
wait	wait	Wait for a process to end

* These routines are available only on CONVEX C Series machines.

The `system` utility

The `system` utility calls operating system utilities (for both ConvexOS and SPP-UX) that are not included in Table 26. The `system` utility executes an executable program, shell script or command and is used as follows:

```
INTEGER FUNCTION SYSTEM (string)
CHARACTER*(*) string
```

The *string* is passed to your shell and executed as a command. The following statement passes the shell a command to rename `FILE1` to `FILE2`:

```
I = SYSTEM ('mv FILE1 FILE2')
```

The following program obtains the exit status of a script called by a CONVEX Fortran program.

```
PROGRAM GET_STATUS
INTEGER*1 STATUS(4)
INTEGER I,SYSTEM
EQUIVALENCE (I,STATUS)

I = SYSTEM ('SHELL_SCRIPT')
WRITE (*,*) 'EXIT STATUS OF SHELL_SCRIPT:',STATUS(3)
END
```

Assume that `SHELL_SCRIPT` associated with the preceding example is:

```
#
ECHO "NOW EXECUTING SCRIPT"
EXIT(66)
```

When the program executes, the following output is produced:

```
NOW EXECUTING SCRIPT
EXIT STATUS OF SHELL_SCRIPT: 66
```

For further information, refer to the system (3F) man page.

VAX-11 Fortran system utilities

These utilities are provided for VAX compatibility. If your program currently calls a VAX/VMS system service, you must change it to call one of the utilities listed in Table 26 or one of the functions listed in the following subsections.

date

The `date` utility returns the current date as *dd-mmm-yy*.

```
SUBROUTINE date(buf)
CHAR*9 buf
```

idate

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 such as 1969.

```
SUBROUTINE idate(iarray)
INTEGER*4 iarray(3)
```

errsns

The `errsns` utility is similar to the function `ierrno` and returns information about the last runtime error.

```
SUBROUTINE errsns(fnum,rmssts,rmsstv,iunit,condval)
INTEGER*4 fnum,rmssts,rmsstv,iunit,condval
```

fnum is the most recent Fortran runtime error number. The remaining arguments are not used.

exit

The `exit` utility ends a process and makes the argument status available to the parent process. It is equivalent to the utility function of the same name.

```
SUBROUTINE exit(status)
INTEGER*4 status
```

secnds

The secnds utility returns the system time in seconds, less the value of its argument.

```
FUNCTION secnds(x)
REAL*4 x
```

time

The time utility returns the current system time in an ASCII string as *hh:mm:ss*.

```
SUBROUTINE time(buf)
CHAR*8 buf
```

ran

The ran utility returns random values. It is similar to the ConvexOS utility rand except that it returns real values in the range 0.0 through 1.0.

```
FUNCTION ran(i)
INTEGER*4 i
```

mvbits

The mvbits utility transfers *len* bits from positions *i* through *i+len-1* of the source location, *m*, to positions *j* through *j+len-1* of the destination location, *n*. The values of *i+len* and *j+len* must be less than or equal to 32.

```
SUBROUTINE mvbits(m, i, len, n, j)
INTEGER*4 m, i, len, n, j
```

When mvbits is called from a program compiled with the -p8, -pd8, or -cfc options, its arguments must all be of type INTEGER*8 and the values of *i+len* and *j+len* must be less than or equal to 64.

Runtime errors and exceptions

6

This chapter discusses runtime error processing and describes how the runtime library processes errors, what the defaults are, and how to override the defaults.

The runtime system software modules support Fortran features that are not handled by the compiler. The modules that make up the runtime system are packaged in precompiled files called libraries, which are accessible by the CONVEX loader, `ld`.

At runtime, error or exception conditions can occur during I/O operations, from system-detected errors, from invalid input data, from arithmetic errors, or from argument errors in calls to the mathematical library. The runtime library provides default processing for errors, sends the appropriate messages, and takes steps to recover from errors, if possible. To override default actions, use:

- `ERR` (error) and `END` (end-of-file) specifiers in I/O statements to transfer control to error-handling code within the program
- `IOSTAT` (I/O status) specifier in I/O statements to identify Fortran- specific errors based on the values of `IOSTAT`
- CONVEX signal-handling facility to modify error processing

I/O error processing

When an I/O error occurs during program execution, the runtime default action is to print an error message and end the program with a core dump. Error numbers lower than 100 are

generated by the operating system, either ConvexOS or SPP-UX. These errors are documented in the intro(2) man page.

ERR and END specifiers

To override program termination on detection of an I/O error, use the ERR or END specifier in I/O statements to transfer control to a specified point in the program. Execution continues at the specified statement and no error message prints. For example, consider this statement:

```
WRITE (8,50,ERR=400)
```

If an error occurs during its execution, the runtime library transfers control to the statement at label 400. Similarly, the END specifier handles an end-of-file condition that otherwise might be treated as an error, as in this example:

```
READ (12,70,END=550)
```

ERR can also be specified as a keyword in an OPEN, CLOSE, or INQUIRE statement:

```
OPEN(UNIT=10,FILE='FILNAM',STATUS='OLD',ERR=999)
```

Detection of an error while this statement is executing transfers control to statement 999.

IOSTAT specifier

To continue program execution after an I/O error and return data on I/O operations, use the IOSTAT specifier. This specifier can augment or replace the END and ERR transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses printing of an error message and defines the specified integer variable or integer array element as:

- A value of -1 when an end-of-file condition occurs
- A value of 0 when no error condition or end-of-file condition occurs
- A positive integer value when an error condition occurs. This value is one of the system errors or Fortran I/O errors. For more information on system and I/O errors, refer to the "Runtime error messages" section of Appendix C, "Compiler and runtime messages."

After the I/O statement executes and an IOSTAT is assigned a value, control transfers to the END or ERR statement label, if one exists. When no control transfer occurs, normal execution continues.

Example:

```
     READ  (5,*,IOSTAT=IERR,ERR=10,END=20)I,J,K
                .
                .
                .
     (process input record)
                .
                .
                .
 10  PRINT *,'ERROR DURING READ:',IERR
     STOP
 20  PRINT *,'END OF FILE'
     STOP
```

When an error occurs in this example, IERR takes the value of the error code, and an error message including the code is printed at line 10.

Signals and exceptions

This section describes the signals and exceptions that can occur at runtime.

Signals

A signal is generated by an abnormal event, a user at a terminal (quit, interrupt, stop), a program error (for example, bus error), the request of another program (kill), or when a process is stopped to access its control terminal while the process is in background mode. Signals can also be generated when a process resumes after being stopped, when the status of a child process changes, or when input is ready at the control terminal.

Table 27 and Table 28 list information about each runtime signal. Each signal has a default action associated with it. For all signals in Table 27 and Table 28 except those with footnotes, the default action is to end the program. The signal routine allows this default action to be overridden for all signals except SIGKILL and SIGSTOP (on C Series) and _SIGKILL and _SIGSTOP (on SPP Series).

The following table lists the name, number, and meaning for CONVEX C Series runtime signals.

Table 27 C Series signal names and numbers (C Series only)

Signal name	No.	Meaning
SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating-point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16**	Urgent condition present on socket
SIGSTOP	17***	Stop (cannot be caught or ignored)
SIGTSTP	18***	Stop signal generated from keyboard
SIGCONT	19**	Continue after stop
SIGCHLD	20**	Child status has changed
SIGTTIN	21***	Background read attempted from control terminal
SIGTTOU	22***	Background write attempted to control terminal
SIGIO	23**	I/O is possible on a descriptor
SIGXCPU	24	CPU time limit exceeded
SIGXFSZ	25	File size limit exceeded
SIGVTALRM	26	Virtual time alarm
SIGPROF	27	Profiling timer alarm
SIGWINCH	28	Window changed
SIGLOST	29	Resource lost
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2

* The default action of these signals is to end the program and produce a core dump.
** The default action of these signals is to ignore the signal.
*** The default action of these signals is to stop the program.

The following table lists the name, number, and meaning for CONVEX SPP Series runtime signals.

Table 28 SPP Series signal names and numbers (SPP Series only)

Signal name	No.	Meaning
_SIGHUP	1	Floating point exception
SIGINT	2	Interrupt
_SIGQUIT	3	Quit
SIGILL	4	Illegal instruction (not reset when caught)
_SIGTRAP	5	Trace trap (not reset when caught)
SIGABRT	6	Process abort signal
_SIGIOT	SIGABRT	IOT instruction
_SIGEMT	7	EMT instruction
SIGFPE	8	Floating point exception
_SIGKILL	9	Kill (cannot be caught of ignored)
_SIGBUS	10	Bus error
SIGSEGV	11	Segmentation violation
_SIGSYS	12	Bad argument to system call
_SIGPIPE	13	Write on a pipe with no one to read it
_SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal from kill
_SIGUSR1	16	User defined signal 1
_SIGUSR2	17	User defined signal 2
_SIGCHLD	18	Child process terminated or stopped
_SIGCLD	_SIGCHLD	Death of a child
_SIGPWR	19	Power state indication
_SIGVTALRM	20	Virtual timer alarm
_SIGPROF	21	Profiling timer alarm
_SIGIO	22	Asynchronous I/O
_SIGPOLL	_SIGIO	Used for HP-UX hpstreams signal
_SIGWINCH	23	Window size change signal
_SIGWINDOW	_SIGWINCH	Exists for compatibility reasons
_SIGSTOP	24	Stop signal (cannot be caught or ignored)
_SIGTSTP	25	Interactive stop signal
_SIGCONT	26	Continue if stopped
_SIGTTIN	27	Read from control terminal attempted by a member of a background process group
_SIGTTOU	28	Write to control terminal attempted by a member of a background process group

Table 28 **(continued)** SPP Series signal names and numbers (SPP Series only)

Signal name	No.	Meaning
_SIGURG	29	Urgent condition on IO channel
_SIGLOST	30	Remote lock lost (NFS)
_SIGRESERVE	31	Saved for future use
_SIGDIL	32	DIL signal

Exceptions

An exception is an event that disrupts the running of a program. Exceptions occur because of problems in the currently executing program (for example, arithmetic inconsistencies or address translation faults), or as a result of some asynchronous event (for example, an interrupt or hardware failure). Exceptions result in the transfer of control to a predetermined address known as an exception or signal handler.

Table 29 shows the C Series mapping of exceptions to signals and codes.

Table 29 C Series mapping of exceptions to signals and codes (C Series only)

Hardware	Signal	Code
Arithmetic traps	SIGFPE (8)	
Integer overflow	SIGFPE	FPE_INTOVF_TRAP (1)
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP (2)
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP (3)
Floating division by zero	SIGFPE	FPE_FLTDIV_TRAP (4)
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP (5)
Reserved operand trap	SIGFPE	FPE_RESOP_TRAP (6)
Segmentation violations	SIGSEGV (11)	
Read access violation	SIGSEGV	SEG_READ_TRAP (1)
Write access violation	SIGSEGV	SEG_WRITE_TRAP (2)
Execute access violation	SIGSEGV	SEG_EXEC_TRAP (3)
Invalid segment	SIGSEGV	SEG_INVSDR_TRAP (4)
Invalid page table page	SIGSEGV	SEG_INVPTP_TRAP (5)
Invalid memory reference	SIGSEGV	SEG_INVDATA_TRAP (6)
I/O access violation	SIGSEGV	SEG_IOACC_TRAP (7)
Ring violations	SIGBUS (10)	
Inward address reference	SIGBUS	BUS_INWADDR_TRAP (1)
Outward ring call	SIGBUS	BUS_OUTCALL_TRAP (2)
Inward ring return	SIGBUS	BUS_INWRTN_TRAP (3)
Invalid syscall gate	SIGBUS	BUS_INVGATE_TRAP (4)
Invalid return frame length	SIGBUS	BUS_INVFRL_TRAP (5)

Table 29 (continued) C Series mapping of exceptions to signals and codes (C Series only)

Hardware	Signal	Code
Illegal instruction	`SIGILL (4)`	
Error exit instruction	`SIGILL`	`ILL_ERRXIT_TRAP (1)`
Privileged instruction	`SIGILL`	`ILL_PRIVIN_TRAP (2)`
Undefined op code	`SIGILL`	`ILL_UNDFOP_TRAP (4)`
Trace pending	`SIGTRAP (5)`	
Bpt instruction	`SIGTRAP (5)`	

Table 30 lists SPP Series mapping of exceptions to signals and codes.

Table 30 SPP Series mapping of exceptions to signals and codes (SPP Series only)

Hardware	Signal	Code
Arithmetic traps	`SIGFPE (8)`	
Integer overflow	`SIGFPE`	`FPE_INTOVF_TRAP (1)`
Integer divide by zero	`SIGFPE`	`FPE_INTDIV_TRAP (2)`
Floating overflow	`SIGFPE`	`FPE_FLTOVF_TRAP (3)`
Floating divide by zero	`SIGFPE`	`FPE_FLTDIV_TRAP (4)`
Floating underflow	`SIGFPE`	`FPE_FLTUND_TRAP (5)`
Decimal overflow	`SIGFPE`	`FPE_DECOVF_TRAP (6)`
Subscript out of range	`SIGFPE`	`FPE_SUBRNG_TRAP (7)`
Floating overflow fault	`SIGFPE`	`FPE_FLTOVF_FAULT (8)`
Floating divide by zero fault	`SIGFPE`	`FPE_FLTDIV_FAULT (9)`
Floating underflow fault	`SIGFPE`	`FPE_FLTUND_FAULT (10)`
Illegal instructions	`SIGILL (4)`	
Reserved addressing fault	`SIGILL`	`ILL_RESAD_FAULT (0)`
Privileged instruction fault	`SIGILL`	`ILL_PRIVIN_FAULT (1)`
Reserved operand fault	`SIGILL`	`ILL_RESOP_FAULT (2)`

Error-processing routines

This section describes general error-processing routines you can use to attach your own signal handler, enable or disable certain arithmetic traps, and retrieve error numbers. The error-processing routines are located in `/usr/lib/libU77.a` and are described in the section 3F of the man pages.

`setjmp` and `longjmp`

The `setjmp` and `longjmp` routines save and restore the stack environment and the signal mask (`sigmask`), respectively, and can be used in processing errors and interrupts encountered in a

low-level subroutine. These routines provide a mechanism for performing statement-level recovery from errors.

Example:

```
INTEGER*4 ENV(10), VAL
I = SETJMP(ENV)
.
.
.
CALL LONGJMP(ENV,VAL)
```

The first time setjmp is called, it returns a value of 0; thereafter, it returns the value of the second argument to longjmp.

Note

Always compile routines that call setjmp and longjmp at optimization level -no. Optimizing these routines may cause unexpected results.

The _setjmp and _longjmp routines save and restore the stack and registers but not the signal mask (sigmask). You cannot use longjmp to restore the environment saved by _setjmp and you cannot use _longjmp to restore the environment saved by setjmp.

errtrap (C Series only)

The errtrap routine enables or disables signal trapping. When signal trapping is enabled, errtrap traps these signals:

- Integer overflow
- Floating-point underflow
- Intrinsic errors
- Integer divide by zero
- Floating-point divide by zero
- Floating-point overflow
- Reserved operand fault

Note

Always compile routines that call errtrap at optimization level -no. Optimizing these routines can cause unexpected results.

The errtrap routine enables or disables trapping for the designated errors by setting or resetting the appropriate bits in the process status word. If trapping is enabled for a particular error and that error occurs, signal SIGFPE is sent to the process. On

completion of the routine calling `errtrap`, the previous value of the flags is restored.

Note

By default, error trapping for integer overflow, floating underflow, and intrinsic routine errors is set to OFF. Set them to ON for debugging.

The argument to `errtrap` is produced by summing the appropriate flags from Table 31.

Table 31 `errtrap` argument flags

Flag	Meaning	Default
'01'X	Trap integer overflow	OFF
'02'X	Trap floating underflow	OFF
'04'X	Trap intrinsic routine errors	OFF
'08'X	Trap floating point (divide by zero, overflow, reserved operand fault)	ON
'10'X	Trap integer divide by zero	ON

By default (if you do not use `errtrap`), these error traps are enabled during execution: integer divide by zero, floating-point overflow, floating-point divide by zero, and reserved operand fault. By default, these traps are disabled during execution: floating-point underflow, integer overflow, and intrinsic routine errors.

The `errtrap` routine supersedes `traper`, which is maintained for upward compatibility. If `traper` is used in a routine that might cause an exception, you must compile the routine at `-no` to insure that `traper` will catch the exception.

```
I = ERRTRAP('3'X)
```

This statement enables integer overflow and floating-point underflow and disables intrinsic routine errors, integer divide by zero, and floating-point trap.

Hardware differences

The `errtrap` routine sometimes causes different messages to be issued for the same errors on different hardware platforms. This is because many math functions that are implemented as library

routines on C1 architectures are implemented in microcode or hardware or both on C2, C3, and C4 architectures. Table 32 enumerates these functions and their purpose. For more C Series information refer to Chapter 2 of the *CONVEX Assembly Language Reference Manual (C Series)*.

Table 32 Intrinsic instructions—C200, C3200, C3400, C3800, and C4600 Series

Instruction mnemonic	Description
`atan.d Sk`	Arctangent of a double-precision number
`atan.s Sk`	Arctangent of a single-precision number
`cos.d Sk`	Cosine of a double-precision number
`cos.s Sk`	Cosine of a single-precision number
`exp.d Sk`	Exponent of a double-precision number
`exp.s Sk`	Exponent of a single-precision number
`sin.d Sk`	Sine of a double-precision number
`sin.s Sk`	Sine of a single-precision number
`sqrt.d Sk`	Square root of a double-precision number
`sqrt.s Sk`	Square root of a single-precision number
`sqrt.d Vj,Vk`	Square root double vector/vector
`sqrt.d.f Vj,Vk`	Square root double using not VM
`sqrt.d.t Vj,Vk`	Square root double using VM
`sqrt.s Vj,Vk`	Square root single vector/vector
`sqrt.s.f Vj,Vk`	Square root single using not VM
`sqrt.s.t Vj,Vk`	Square root single using VM

As an example of the different messages issued for trying to take the square root of a negative number, consider the following program:

```
PROGRAM EXAMPLE
I = ERRTRAP('4'X)
Y = -1.
X = SQRT(Y)
PRINT *, X,Y
END
```

Figure 14 shows the errors reported when the program is compiled and run on a CONVEX C1 Series.

Figure 14 C1 Series intrinsic errors

```
% a.out
mth$r_sqrt: [300] square root undefined for negative values
*** IOT Trap: at 445d0942

_gen$soff+21280000(6) from 8002f484 [ap = ffffca5c]
_abort() from 80029d1a [ap = ffffca84]
_mth$err() from 80029b4c [ap = 80048000]
_MAIN__() from 800017d4 [ap = ffffcaac]
_main(1,ffffcafc,ffffcb04) from 800010d0 [ap = ffffcaf0]
IOT trap (core dumped)
%
```

Figure 15 shows the errors reported when the program is compiled and run on a CONVEX C2 Series system. Errors reported for different C2, C3, or C4 Series architectures are similar.

Figure 15 C2, C3, and C4 Series intrinsic errors

```
% a.out
Intrinsic instruction: [300] square root undefined for negative values
_mth$err(12c,800374e9) from 80001662 [ap = 8004ae40]
_main+1b2(8004aeb8) from 800016cc [ap = 8004ae6c]
signal(8,10,8004aeb8,80001694,80001372) from ffffd0a6 [ap = 8004aea4]
mth$hwtype+c(800013a8) from 80001372 [ap = 800013b0]
_MAIN__() from 800015fc [ap = ffffcabc]
_main(1,ffffcb10,ffffcb18) from 800010d0 [ap = ffffcb04]
IOT trap (core dumped)
%
```

In both cases error [300] was flagged; however, it was described by its runtime function name on the C1 (`mth$r_sqrt`), whereas on the C2 it was listed as an intrinsic routine. Other functions listed in the table similarly yield different messages on different machines when `errtrap` is used to flag errors for them.

signal

The `signal` routine designates a signal-handling routine. The `signal` routine has three parameters: the signal number, the condition handler, and a flag. The legal values of the flag are -1, 0, and 1.

```
I = SIGNAL(18, SIGDIE,-1)
```

This statement establishes the condition handler, `SIGDIE`, for stop signals generated from the keyboard.

```
I = SIGNAL(18,0,0)
```

This statement restores the default action for stop signals generated from the keyboard.

```
I = SIGNAL(18,0,1)
```

This statement causes stop signals generated from the keyboard to be ignored.

Note

Always compile routines that call `signal` at optimization level `-no`. Optimizing these routines can cause unexpected results.

traceback

The `traceback` routine prints the names of the routines in the call stack. Control then returns to the calling program.

Figure 16 shows the output of the `traceback` routine as the result of a floating point exception error.

Figure 16 `traceback` resulting from an exception

```
*** Floating Point Exception: Floating divide by zero: at 800010d8.

signal(8,4,8002bf84,80001262) from ffffd084 [ap = 8002bf74]
curbrk+6d0(80018000,80018004) from 800010d8 [ap = 800010fc]
_MAIN__() from 80001230 [ap = ffffce54]
_main(1,ffffce98,ffffcea0) from 80001074 [ap = ffffce8c]
```

Figure 17 shows the output of the traceback routine generated by a user-called subroutine.

Figure 17 traceback generated by a user-called subroutine

```
_sub2_(80001148,80001154) from 800010de [ap = ffffcde0]
_sub1_(80001148) from 800010ba [ap = 80001150]
_MAIN__() from 800012f4 [ap = ffffce1c]
_main(1,ffffce60,ffffce68) from 80001074 [ap = ffffce54]
```

The perror, gerror, and ierrno routines

The perror, gerror, and ierrno routines retrieve the system error message numbers. perror writes a message appropriate to the last detected system error to stderr (on C Series unit 0, on SPP Series unit 7). gerror returns the system error message in a character variable and can be called either as a subroutine or as a function. ierrno returns the error number of the last detected system error, which is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call that indicates what caused the error.

For more information about these routines, consult the perror(3F) man page.

Examples of signal handling

Figure 18 illustrates handling signals from interrupts.

Figure 18 Signal handler for interrupts

```
C THIS EXAMPLE ESTABLISHES A SIGNAL HANDLER FOR INTERRUPTS
C
       INTEGER SIGNAL       ! INTEGER FUNCTION
       INTEGER OLDHANDLER   ! SAVE OLD SIGNAL VALUE
       INTEGER NEWHANDLER   ! NEW HANDLER ADDRESS
       EXTERNAL NEWHANDLER
       OLDHANDLER = SIGNAL (2, NEWHANDLER, -1) ! ENABLE SIGNAL HANDLER
       PRINT *,'HIT CONTROL-C (^C) TO GENERATE A SIGINT SIGNAL...'
       SLEEP(999999)        ! WAIT HERE UNTIL USER ENTERS ^C
       END

C SUBROUTINE TO INTERCEPT SIGNALS
C
       SUBROUTINE NEWHANDLER (SIG, CODE, SCP)
       INTEGER SIG          ! SIGNAL NUMBER
       INTEGER CODE         ! SIGNAL SUBCODE
       INTEGER SCP(5)       ! SIGNAL CONTEXT
                            ! (1) /* SIGSTACK STATE TO RESTORE */
                            ! (2) /* SIGNAL MASK TO RESTORE */
                            ! (3) /* SP TO RESTORE */
                            ! (4) /* PC TO RESTORE */
                            ! (5) /* PSW TO RESTORE */
       WRITE (*,00002) SIG,CODE,SCP(4)
00002  FORMAT(/,
     $         ' SIGNAL NUMBER [SIGINT]....:',I10,/,
     $         ' SIGNAL LSUBCODE [0].......:',I10,/,
     $         ' PROGRAM COUNTER [PC]......:',Z10,/,
     $         ' --------------------------------')
       END
```

Figure 19 illustrates handling signals from arithmetic exceptions.

Figure 19 Signal handler for arithmetic exceptions

```
      PROGRAM PR3527F
      EXTERNAL SIGHANDLER
      INTEGER*4 ENV(10),I,CODE,SETJMP,LONGJMP
      COMMON ENV

C ESTABLISH A SIGNAL HANDLER FOR ARITHMETIC EXCEPTIONS
      OLDHAN = SIGNAL(8,SIGHANDLER,-1)

C ESTABLISH AN ENVIRONMENT FOR RECOVERY IN CASE AN ERROR OCCURS
C WITHIN THE ROUTINE WORK. IN CASE OF ERROR, ROUTINE FIXUP IS
C CALLED TO REPAIR THE PROGRAM STATE SO EXECUTION MAY CONTINUE.
      I = SETJMP(ENV)

C INITIALLY, SETJMP RETURNS 0. IF A SUBSEQUENT LONGJMP IS PERFORMED,
C THE VALUE RETURNED IS THE SECOND ARGUMENT TO LONGJMP.
C RETURNING A VALUE OF ZERO IS NOT RECOMMENDED.
      IF (I .EQ. 0) THEN
        CALL WORK ()
      ELSE
        CALL FIXUP ()
      ENDIF
      ...
      END

      SUBROUTINE SIGHANDLER (SIG,CODE,SCP)
C INTERCEPT (SIGFPE) FLOATING POINT EXCEPTIONS
      INTEGER*4 SIG,CODE,SCP(5),ENV(10)
      COMMON ENV

C RETURN THE ERROR SUBCODE AND EXECUTE GLOBAL GOTO
      CALL LONGJMP(ENV,CODE)
      END

      SUBROUTINE WORK
      INTEGER A,B,C
      PRINT *,'DOING MEANINGFUL WORK.'
      READ *,A,B,C
      A = B/C
      END

      SUBROUTINE FIXUP
      PRINT *,'FIXING RESULTS.'
      END
```

Compiler directives

A

Compiler directives

This appendix briefly describes the compiler directives that are available in CONVEX Fortran on both CONVEX C Series and CONVEX SPP Series machines.

Some compiler directives provide information to the compiler that it cannot determine on its own. Other directives instruct the compiler to override certain default conditions that control optimization, parallelization, or (on C Series machines) vectorization.

Using compiler directives

In CONVEX Fortran, a compiler directive line has the following format:

```
C$DIR [SPP | CSERIES] directive [, directive...]
```

A directive line begins in column one with the characters `C$DIR`. If one of the optional target machine attributes (`SPP` or `CSERIES`) is specified, the directive line will apply only when compiled for the target machine (either CONVEX SPP Series or CONVEX C Series). Following the optional target machine is one or more of the directives described in this appendix. If two or more directives are specified, they are separated by commas. A directive must fit on one line; it cannot be continued. A directive can be surrounded by any number of comment lines.

Table 33 lists the compiler directives that may be specified on CONVEX C Series machines and CONVEX SPP Series machines.

Table 33 Compiler directives available on C Series and SPP Series machines

C Series and SPP Series	C Series only	SPP Series only
BEGIN_TASKS	FORCE_PARALLEL	BARRIER
END_TASKS	FORCE_PARALLEL_EXT	BLOCK_LOOP
LOOP_PRIVATE	FORCE_VECTOR	BLOCK_SHARED
NEXT_TASK	MAX_TRIPS	CRITICAL_SECTION
NO_PARALLEL	NO_RECURRENCE	END_CRITICAL_SECTION
NO_PEEL	NO_VECTOR	END_ORDERED_SECTION
NO_PROMOTE_TEST	PREFER_PARALLEL_EXT	FAR_SHARED
NO_SIDE_EFFECTS	PREFER_VECTOR	FAR_SHARED_POINTER
NO_UNROLL_AND_JAM	PSTRIP	GATE
PEEL	SELECT	LOOP_PARALLEL
PEEL_ALL	SYNCH_PARALLEL	NEAR_SHARED
PREFER_PARALLEL	VSTRIP	NEAR_SHARED_POINTER
PROMOTE_TEST		NO_BLOCK_LOOP
PROMOTE_TEST_ALL		NODE_PRIVATE
ROW_WISE		NODE_PRIVATE_POINTER
SCALAR		NO_LOOP_DEPENDENCE
TASK_PRIVATE		ORDERED_SECTION
UNROLL		SAVE_LAST
UNROLL_AND_JAM		THREAD_PRIVATE
		THREAD_PRIVATE_POINTER

A directive associated with a loop affects the loop that immediately follows the directive and does not affect loops nested within that loop. When using directives on loops, remember that loops can be executed in the four following ways:

- Serial
- Vector but not parallel (only on CONVEX C Series machines)
- Parallel but not vector
- Parallel outer strip and vector inner strip (only on C Series machines)

Descriptions of compiler directives

The remaining sections in this appendix describe the compiler directives available in CONVEX Fortran on C Series and SPP Series machines. Some directives are available only on one series of machine or the other and are marked in the text to indicate this.

BARRIER (SPP Series only)

The BARRIER compiler directive declares one or more variables of data type BARRIER. This directive is useful only in code compiled at optimization level -O3. The BARRIER directive has the form

```
C$DIR BARRIER(barr-name [, barr-name...])
```

where

barr-name is the name of the BARRIER variable to be declared.

By using BARRIER variables and the supporting intrinsic routines, programmers can synchronize threads at specified points in a program. For more information about barriers, refer to Chapter 12 of the *Fortran Language Reference*.

BEGIN_TASKS, NEXT_TASK, END_TASKS

The BEGIN_TASKS set of directives identifies a group of tasks for independent, parallel execution. Using these tasking directives, you can parallelize code outside of loops. These directives are effective only in code compiled at optimization level -O3.

A task is a section of code that can be executed in parallel with other tasks. In CONVEX Fortran, a group of tasks begins with a BEGIN_TASKS directive and ends with an END_TASKS directive. A NEXT_TASK directive precedes the second task and all subsequent tasks in the group.

The BEGIN_TASKS directive has the form

```
C$DIR BEGIN_TASKS [(attribute-list)]
```

where the optional *attribute-list* qualifies the way in which tasks are run in parallel. The ORDERED attribute is available on C Series machines and additional attributes can be specified on SPP Series machines.

The ORDERED attribute, on both C Series and SPP Series machines, specifies that all tasks should be initiated in lexical order; this forces the compiler to begin the first task in the sequence on its respective thread before the second and so on. In absence of the ORDERED attribute, the starting order is indeterminate. Although ORDERED ensures an ordered starting sequence, it does not provide synchronization between tasks and does not guarantee any particular ending order.

SPP Series only

On CONVEX SPP Series machines, *attribute-list* can be any one of the following combinations of attributes:

- ORDERED
- NODES
- THREADS
- MAX_THREADS=*m*
- ORDERED, NODES
- ORDERED, THREADS
- ORDERED, MAX_THREADS=*m*
- NODES, MAX_THREADS=*m*
- THREADS, MAX_THREADS=*m*
- ORDERED, NODES, MAX_THREADS=*m*
- ORDERED, THREADS, MAX_THREADS=*m*

The ORDERED attribute on SPP Series machines behaves as described earlier in this section.

On SPP Series machines, the NODES attribute causes the tasks to run node-parallel, on one thread per available hypernode. The THREADS attribute causes the tasks to run thread-parallel, and is the default.

The ORDERED,NODES and ORDERED,THREADS attributes are available on SPP Series machines and cause the tasks to run ordered node-parallel and ordered thread-parallel, respectively.

Also on SPP Series machines, attributes that specify MAX_THREADS=*m* will run no more than *m* threads, where *m* is an integer constant whose value is known at compile time.

Caution

On both C Series and SPP Series machines, if the task contains a subroutine call and a variable passed to the subroutine is referenced within the task, the compiler issues a warning and fails to parallelize the task. If possible, move the variable reference to before the BEGIN_TASKS directive to allow parallelization.

For more information on tasking directives, see also the "TASK_PRIVATE" directive in this chapter. The following code illustrates the use of the tasking directives. The code on the left is equivalent to the loop on the right.

Tasking code

```
C$DIR BEGIN_TASKS
      statement-1
      ...
C$DIR NEXT_TASK
      statement-2
      ...
C$DIR NEXT_TASK
      statement-3
      ...
C$DIR END_TASKS
```

Loop code

```
C$DIR FORCE_PARALLEL
      DO 100 I = 1,3
         GOTO(10,20,30),I
 10      statement-1
         ...
         GOTO 100
 20      statement-2
         ...
         GOTO 100
 30      statement-3
         ...
 100  CONTINUE
```

BLOCK_LOOP (SPP Series only)

The BLOCK_LOOP compiler directive advises the compiler to use a specified block factor to strip mine the immediately following loop. This directive takes the form

```
C$DIR BLOCK_LOOP [ (BLOCK_FACTOR = n) ]
```

where *n* is the blocking factor requested for strip mining. If the block factor is omitted, the compiler selects an appropriate loop blocking factor. Loop blocking and strip mining are described in detail in the *Exemplar Programming Guide.*

BLOCK_LOOP is effective only at optimization levels -O2 and -O3, when loop blocking may occur.

BLOCK_SHARED (SPP Series only)

Declares the specified allocatable arrays as being of memory type block-shared. Block-shared arrays are distributed equally

among all hypernodes on which a program is running; pages of the array are distributed in same-size blocks across all the subcomplex's hypernodes to accomplish this.

The BLOCK_SHARED directive is available only on SPP Series machines; it takes the form

```
C$DIR BLOCK_SHARED(alloc-arr [, alloc-arr...])
```

where *alloc-arr* is an allocatable array that appears in a prior ALLOCATABLE statement. Multiple arrays may be declared BLOCK_SHARED in one statement.

If a block-shared array's size is not an integral multiple of the page size × num_nodes() then the compiler automatically increases the array's allocation to meet this criterion. The LOOP_PRIVATE directive is equivalent to BLOCK_SHARED on SPP Series machines. For more information see the *Exemplar Programming Guide*.

CRITICAL_SECTION, END_CRITICAL_SECTION (SPP Series only)

The CRITICAL_SECTION and END_CRITICAL_SECTION directives mark a section of code that must be executed by one thread at a time. This directive has the form

```
C$DIR CRITICAL_SECTION[(gate-var)]
```

where the optional *gate-var* is a previously declared GATE variable to be used to control execution of the designated section of code.

The END_CRITICAL_SECTION directive specifies the point where the critical section ends. It does not accept a GATE variable and has the form

```
C$DIR END_CRITICAL_SECTION
```

FAR_SHARED (SPP Series only)

The FAR_SHARED directive instructs the compiler to store a specified list of variables in far-shared memory.

The FAR_SHARED directive is available only on CONVEX SPP Series machines.

This directive uses the following format:

```
C$DIR FAR_SHARED (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names. Variables listed in *namelist* are stored in far-shared memory rather than shared memory (the default). For more information about memory types available on CONVEX SPP Series machines, refer to the *Exemplar Programming Guide*.

Compiler directives

FAR_SHARED_POINTER (SPP Series only)

The FAR_SHARED_POINTER directive places a hidden, compiler-generated pointer to the specified variable in far-shared memory, regardless of the memory class to which the variable is allocated. This directive is available only on SPP Series machines; it has the form

```
C$DIR FAR_SHARED_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

Additional directives, THREAD_PRIVATE_POINTER, NODE_PRIVATE_POINTER, and NEAR_SHARED_POINTER, place pointers in the other classes of memory.

For detailed information about writing programs that optimally use the memory classes available on CONVEX SPP Series machines, consult the *Exemplar Programming Guide*.

FORCE_PARALLEL (C Series only)

The FORCE_PARALLEL directive tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. This directive is effective only in code compiled at optimization level -O3.

FORCE_PARALLEL is available only on CONVEX C Series machines.

Certain actual dependencies, such as from one scalar to another, can cause the compiler to ignore this directive. You can use this directive on a loop whether or not the loop contains calls, but it may not be safe to do so.

FORCE_PARALLEL does not allow interchange or distribution of outer loops for vectorization. To enable those optimizations, use FORCE_PARALLEL_EXT.

Caution

This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you use this directive with the SCALAR or NO_RECURRENCE directive, a warning is issued. In addition, an error occurs when you use FORCE_PARALLEL and another parallelizing directive in the same loop nest.

Example:

```
C$DIR FORCE_PARALLEL
      DO I = 1, N
      ENDDO
```

FORCE_PARALLEL_EXT (C Series only)

The FORCE_PARALLEL_EXT directive forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. Loops can be parallelized with FORCE_PARALLEL_EXT whether or not they contain calls. This directive is effective only in code compiled at optimization level -O3.

FORCE_PARALLEL_EXT is available only on CONVEX C Series machines. On SPP Series machines, the LOOP_PARALLEL directive is equivalent to FORCE_PARALLEL_EXT.

If FORCE_PARALLEL_EXT and the FORCE_VECTOR directive are specified for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

FORCE_PARALLEL_EXT allows interchange of outer loops for vectorization.

Caution

This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you attempt to use this directive with the SCALAR or NO_RECURRENCE directive, an error occurs. In addition, an error occurs when you try to use FORCE_PARALLEL_EXT and another parallelizing directive in the same loop nest.

FORCE_VECTOR (C Series only)

The FORCE_VECTOR directive forces the compiler to vectorize the loop that follows, regardless of apparent recurrences. It is possible to use a FORCE_VECTOR directive with a loop that would be fully vectorized without the directive and get incorrect answers because the directive causes the compiler to ignore dependencies. This directive is effective only in code compiled at optimization level -O2 or higher.

FORCE_VECTOR is available only on CONVEX C Series machines.

This directive should be used with fully vectorizable loops. If FORCE_VECTOR and FORCE_PARALLEL_EXT are specified for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

Caution

This directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the vectorized code.

A warning is issued if this directive is used with the SCALAR directive or with the NO_RECURRENCE directive. In addition, an error occurs when you attempt to use FORCE_VECTOR and another vectorizing directive in the same loop nest.

GATE (SPP Series only)

The GATE directive declares one or more variables of type GATE. This directive is useful only in code compiled at optimization level -O3. The format of this directive is

```
C$DIR GATE(gate-name [, gate-name...])
```

where

gate-name is the name of the GATE variable to be declared.

By using GATE variables and the attendant intrinsic routines, programmers can limit execution of a block of code to one thread at a time. For more information about GATE variables and intrinsics, see Chapter 12 of the *CONVEX Fortran Language Reference*.

LOOP_PARALLEL (SPP Series only)

The LOOP_PARALLEL directive specifies that the immediately following loop should be run in parallel. This directive is effective only in code compiled at optimization level -O3.

The LOOP_PARALLEL compiler directive is available only on CONVEX SPP Series machines.

A loop marked with a LOOP_PARALLEL directive must have a known number of iterations at loop invocation time. For this reason, LOOP_PARALLEL does *not* apply to WHILE loops, loops containing RETURN or STOP statements, or any other loop that exits by abnormal termination. A LOOP_PARALLEL loop will not be interchanged with any other loop in a loop nest.

The LOOP_PARALLEL directive does not apply to the implicit loops of Fortran 90 array section syntax; Fortran 90 array assignments are data-independent and are candidates for automatic parallelization.

LOOP_PARALLEL differs from the PREFER_PARALLEL directive in that LOOP_PARALLEL parallelizes the loop that follows regardless of any loop-carried dependencies.

Any secondary induction variables in a LOOP_PARALLEL loop must be written as linear expressions of the primary induction variable and must be declared LOOP_PRIVATE. See the "LOOP_PRIVATE" section of this chapter for more information.

The format of the LOOP_PARALLEL directive is as follows:

```
C$DIR LOOP_PARALLEL [(attribute-list)]
```

where the optional *attribute-list* can contain any one of the following combinations of attributes:

- THREADS—causes thread-level parallelism.
- NODES—causes node-level parallelism.
- CHUNK_SIZE = *n*—divides the loop into chunks of *n* iterations and distributes the chunks round-robin to the processors.
- MAX_THREADS = *m*—allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.
- ORDERED—causes ordered execution of the loop; provides no automatic synchronization.

- ORDERED, NODES—causes ordered execution across hypernodes.
- ORDERED, THREADS—causes ordered execution across threads.
- NODES, CHUNK_SIZE = *n*—node-level parallelism by chunks.
- THREADS, CHUNK_SIZE = *n*—thread-level parallelism by chunks.
- CHUNK_SIZE = n, MAX_THREADS = *m*—chunk parallelism on no more than *m* threads.
- ORDERED, MAX_THREADS = *m*—ordered parallelism on no more than *m* threads.
- NODES, MAX_THREADS = *m*—node-level parallelism on no more than *m* hypernodes.
- THREADS, MAX_THREADS = *m*—thread-level parallelism on no more than *m* threads.
- ORDERED, NODES, MAX_THREADS = *m*—ordered node-level parallelism on no more than *m* hypernodes.
- ORDERED, THREADS, MAX_THREADS = *m*—ordered thread-level parallelism on no more than *m* threads.
- NODES, CHUNK_SIZE = *n*, MAX_THREADS = *m*—node-level parallelism by chunks of size *n* on no more than *m* hypernodes
- THREADS, CHUNK_SIZE = *n*, MAX_THREADS = *m*—thread-level parallelism by chunks of size *n* on no more than *m* threads.
- IVAR = *ivarname*—specifies the primary induction variable (*ivarname*) of the loop. Required for DO WHILE and "hand-rolled" loops.

Example usage of the ORDERED and CHUNK_SIZE attributes is provided in the "PREFER_PARALLEL" section of this chapter. For more detailed information about optimally parallelizing code on CONVEX SPP Series machines, consult the *Exemplar Programming Guide.*

Caution

The LOOP_PARALLEL directive causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check results generated by the parallelized code by comparing them to the results generated by the nonparallelized code.

Compiler directives

LOOP_PRIVATE

The LOOP_PRIVATE directive declares a list of variables and/or arrays private to the immediately following DO loop. This directive is effective only in code compiled at optimization level -O3.

The compiler assumes that variables declared LOOP_PRIVATE have no loop-carried dependencies. No starting or ending values can be assumed for these variables.

The LOOP_PRIVATE directive has the form

```
C$DIR LOOP_PRIVATE(varlist)
```

where

varlist

is a list of scalar variables or arrays, separated by commas, that are to be private to the immediately following loop.

Only scalar variables and statically-sized arrays can be declared private. Dynamic, allocatable, and automatic arrays are not allowed. Structures are not allowed.

Including induction variables (i.e. DO loop indices) in *varlist* will yield wrong answers for automatically parallelized loops.

If a variable that appears in *varlist* is referenced in an iteration of the loop, it must have been assigned a value previously on that iteration of the loop. Values assigned outside the loop or in previous iterations will not be available.

If the variable is referenced after the loop, it must have been assigned a value after the loop. Values assigned inside the loop or before the loop are not available.

All shared scalar variables that are assigned within a parallel loop, including nested loop induction variables, must be declared LOOP_PRIVATE, except shared variables that are guarded with locks.

The following example demonstrates use of the LOOP_PRIVATE directive.

```
C$DIR   PREFER_PARALLEL
C$DIR   LOOP_PRIVATE(S)
        DO I=1,N
          IF (I .GT. ILIM) THEN
            S = 3.0
          ELSE IF (I .LE. ILIM) THEN
            S = 2.0
          ENDIF
          A(I) = S
        ENDDO
```

In this example, S must have a value for the assignment to A(I) at the end of the loop. Without the LOOP_PRIVATE directive, the compiler cannot tell that S is always assigned. It therefore assumes that the value of S from a previous iteration might be needed, and fails to parallelize the loop. The presence of LOOP_PRIVATE(S) tells the compiler to ignore the possible dependency on S, so that the PREFER_PARALLEL directive can be honored and the loop can be parallelized.

MAX_TRIPS (C Series only)

The MAX_TRIPS directive instructs the compiler that the following loop is never executed more than the specified number of times. This directive is available only on CONVEX C Series machines and is effective only in code compiled at optimization level -O3.

The format of MAX_TRIPS is

```
C$DIR MAX_TRIPS (n)
```

where the value of *n* is less than or equal to the vector register length of 128.

This directive can be used to prevent strip mining, when it might otherwise be performed. The elimination of strip mining results in more efficient code generation.

NEAR_SHARED (SPP Series only)

The NEAR_SHARED directive instructs the compiler to store a specified list of variables in near-shared memory.

The NEAR_SHARED directive is available only on CONVEX SPP Series machines.

This directive uses the following format:

```
C$DIR NEAR_SHARED (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names. Variables listed in *namelist* are stored in near-shared memory rather than shared memory (the default). For more information about memory types available on CONVEX SPP Series machines, refer to the *Exemplar Programming Guide*.

NEAR_SHARED_POINTER (SPP Series only)

The NEAR_SHARED_POINTER directive places a hidden, compiler-generated pointer to the specified variable in near-shared memory, regardless of the memory class to which the variable is allocated. This directive is available only on SPP Series machines; it has the form

```
C$DIR NEAR_SHARED_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

Additional directives, THREAD_PRIVATE_POINTER, NODE_PRIVATE_POINTER, and FAR_SHARED_POINTER, place pointers in the other classes of memory.

For detailed information about writing programs that optimally use the memory classes available on CONVEX SPP Series machines, consult the *Exemplar Programming Guide*.

NO_BLOCK_LOOP (SPP Series only)

Informs the compiler to perform no blocking on the immediately following loop. Normally, the CONVEX Fortran compiler attempts to determine the best blocking in order to achieve cache data reuse between loop iterations in loop nests.

NO_BLOCK_LOOP is available only on SPP Series machines.

NODE_PRIVATE (SPP Series only)

The NODE_PRIVATE directive instructs the compiler to store a specified list of variables in node-private memory.

The NODE_PRIVATE directive is available only on CONVEX SPP Series machines.

This directive uses the following format:

```
C$DIR NODE_PRIVATE (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names. Variables listed in *namelist* are stored in node-private memory rather than shared memory (the default). For more information about memory types available on CONVEX SPP Series machines, refer to the *Exemplar Programming Guide.*

Compiler directives

NODE_PRIVATE_POINTER (SPP Series only)

The NODE_PRIVATE_POINTER directive places a hidden, compiler-generated pointer to the specified variable in node-private memory, regardless of the memory class to which the variable is allocated. This directive is available only on SPP Series machines; it has the form

```
C$DIR NODE_PRIVATE_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

Additional directives, THREAD_PRIVATE_POINTER, NEAR_SHARED_POINTER, and FAR_SHARED_POINTER, place pointers in the other classes of memory.

For detailed information about writing programs that optimally use the memory classes available on CONVEX SPP Series machines, consult the *Exemplar Programming Guide.*

NO_LOOP_DEPENDENCE (SPP Series only)

The NO_LOOP_DEPENDENCE directive instructs the compiler that it can safely ignore any potential loop-carried dependencies (LCDs) it detects in specified arrays. The NO_LOOP_DEPENDENCE directive is effective only in code compiled at optimization level -O3. This directive has the form

```
C$DIR NO_LOOP_DEPENDENCE (namelist)
```

where

> *namelist* is a list of arrays whose potential LCDs will be ignored by the compiler.

Use NO_LOOP_DEPENDENCE for arrays only. Use LOOP_PRIVATE to specify dependence-free scalar variables.

NO_PARALLEL

The NO_PARALLEL directive tells the compiler not to parallelize the loop that immediately follows; vectorization (available on C Series machines) is not prevented. This directive is effective only in code compiled at optimization level -O2 or higher.

NO_PEEL

The NO_PEEL directive prevents the compiler from applying loop boundary value peeling to the loop that immediately follows. This directive overrides boundary level peeling at all levels: default, -peel, and -peelall. This directive is effective only in code compiled at optimization level -O2 and higher. For information about using this directive on C Series machines, consult the *CONVEX Fortran Optimization Guide* Chapter 3, "Vector optimization."

NO_PROMOTE_TEST

The NO_PROMOTE_TEST directive prevents the compiler from applying test promotion to the loop that immediately follows. This directive overrides test promotion at all levels: default, -ptst, and -ptstall. This directive is effective only in code compiled at optimization level -O2 and higher. For information about using this directive on C Series machines, consult the *CONVEX Fortran Optimization Guide* Chapter 3, "Vector optimization."

NO_RECURRENCE (C Series only)

The NO_RECURRENCE directive instructs the compiler to disregard an apparent recurrence in a loop. If there is no other impediment to vectorization, the loop is vectorized. This directive is effective only in code compiled at optimization level -O2 or higher.

NO_RECURRENCE is available only on CONVEX C Series machines.

The NO_RECURRENCE directive does not affect recurrences caused by a nested DO loop. The directive can, however, be used on each loop in a nest to give the vectorizer maximum opportunity for improving the performance of the nest.

When the NO_RECURRENCE directive is used, the compiler breaks the recurrence by arbitrarily removing one or more dependencies of the cycle.

```
C$DIR NO_RECURRENCE
      DO 10 I + 1,N
10    A(I) = A(I+J)
```

In this example, if J is positive, there is no recurrence.

The compiler always processes a NO_RECURRENCE directive when the apparent recurrence involves an array element. The compiler always ignores a NO_RECURRENCE directive when the apparent recurrence involves a scalar. In the latter case, the compiler knows that a recurrence exists.

Caution

Incorrect results can occur if you mistake a real recurrence for an apparent one. Always test vector results against scalar results to determine whether a recurrence is real or apparent.

For more information on the NO_RECURRENCE directive, refer to Chapter 10, "Limits of optimization."

Compiler directives

NO_SIDE_EFFECTS

The NO_SIDE_EFFECTS directive instructs the compiler that the specified functions *do not* modify the value of a parameter or COMMON variable, perform a read or write, or call another routine that has side effects. This directive is effective at all optimization levels except -no (no optimization).

The format of this directive is:

C$DIR NO_SIDE_EFFECTS (*func* [, *func*...])

where

func specifies one or more user-defined functions.

This directive allows scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that is never used. The function call can be removed because the function has no side effects—it does not matter whether or not the call is made. Such optimization opportunities usually arise after other optimizations are performed and rarely occur in the original source text.

Although the directive can appear anywhere in a program unit, to be effective it must be used before the named function is called. Use the directive if the compiler gives the advisory message `More optimization is possible if this function call has no side effects.` If there are no arguments, the directive applies to all functions referenced (textually) after the directive.

Example:

```
C$DIR NO_SIDE_EFFECTS (F1,F2)
      .
      .
      .
      X=Y* F1(5,Z)-W !IF THE X= DOES NOT REACH
      .              !A USE OF X, THE ASSIGNMENT
      .              !STMT CAN BE REMOVED
      .
```

A function call with no side effects is invariant with respect to a loop under these conditions:

- The function call's arguments do not vary within the loop and the function call can be moved out of the loop.
- The function call does not modify a COMMON variable.
- The function call does not perform I/O.

NO_UNROLL_AND_JAM

Disallows the unroll and jam transformation for the immediately following loop only. On SPP Series machines the unroll and jam transformation is enabled by default. This directive affects optimization levels -O2 and -O3.

For more information about unroll and jam, see also this chapter's section on the UNROLL_AND_JAM directive and coverage of the -uj, -nuj, and -ujn compiler options in Chapter 1. For additional information about the unroll and jam transformation on SPP Series machines, refer to the *Exemplar Programming Guide.*

NO_VECTOR (C Series only)

The NO_VECTOR directive tells the compiler not to vectorize the loop that immediately follows; parallelization is not prevented. This directive is effective only in code compiled at optimization level -O2 or higher.

NO_VECTOR is available only on CONVEX C Series machines.

If the NO_PARALLEL and NO_VECTOR directives both precede a loop, the result is the same as if the SCALAR directive is used.

ORDERED_SECTION, END_ORDERED_SECTION (SPP Series only)

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). This directive is useful only in programs compiled at optimization level -O3.

The ORDERED_SECTION directive has the following form:

```
C$DIR ORDERED_SECTION (gate)
```

where *gate* specifies a previously declared GATE variable 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.

For more information about the ORDERED_SECTION directive, refer to Chapter 12, "SPP Synchronization," of the *Fortran Language Reference*.

PEEL

The PEEL directive allows the compiler to peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound. This directive is effective only in code compiled at optimization level -O2 or higher. Refer to the *CONVEX Fortran Optimization Guide* Chapter 3, "Vector optimization," for information about using this directive on C Series machines.

PEEL_ALL

The PEEL_ALL directive allows the compiler to peel the loop immediately following the directive, expanding the code without bound. This directive is effective only in code compiled at optimization level -O2 or higher. For information about using this directive on C Series machines, refer to the *Fortran Optimization Guide* Chapter 3, "Vector optimization."

PREFER_PARALLEL

The PREFER_PARALLEL directive tells the compiler to parallelize the loop immediately following the directive if it is safe to do so.

The compiler checks first for actual loop-carried dependencies; if none is found, the loop is parallelized. This directive is effective only in code compiled at optimization level -O3.

PREFER_PARALLEL differs from the LOOP_PARALLEL and FORCE_PARALLEL directives in that PREFER_PARALLEL does not parallelize the loop that follows if any loop-carried dependencies exist. LOOP_PARALLEL is available only on SPP Series machines.

A loop marked with a PREFER_PARALLEL directive must have a known number of iterations at loop invocation time. For this reason, PREFER_PARALLEL cannot apply to WHILE loops, loops containing RETURN or STOP statements, or any other loop that exits by abnormal termination. A PREFER_PARALLEL loop will not be interchanged with any other loop in a loop nest.

The PREFER_PARALLEL directive does not apply to the implicit loops of Fortran 90 array section syntax; Fortran 90 array assignments are data-independent and are candidates for automatic parallelization.

The PREFER_PARALLEL directive has the form:

C$DIR PREFER_PARALLEL [(*attribute-list*)]

where the optional *attribute-list* qualifies the way in which the immediately following loop is parallelized.

On CONVEX C Series machines, only the CHUNK_SIZE and ORDERED attributes are available; both these attributes are described later in this section.

SPP Series only

On SPP Series machines, the optional *attribute-list* can contain any one of the following combinations of attributes:

- THREADS—causes thread-level parallelism.
- NODES—causes node-level parallelism.
- CHUNK_SIZE = *n*—divides the loop into chunks of *n* iterations and distributes the chunks round-robin to the processors.
- MAX_THREADS = *m*—allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.
- ORDERED—causes ordered execution of the loop; provides no automatic synchronization.
- ORDERED, NODES—causes ordered execution across hypernodes.

- ORDERED, THREADS—causes ordered execution across threads.
- NODES, CHUNK_SIZE = *n*—node-level parallelism by chunks.
- THREADS, CHUNK_SIZE = *m*—thread-level parallelism by chunks.
- CHUNK_SIZE = n, MAX_THREADS = *m*—chunk parallelism on no more than *m* threads.
- ORDERED, MAX_THREADS = *m*—ordered parallelism on no more than *m* threads.
- NODES, MAX_THREADS = *m*—node-level parallelism on no more than *m* hypernodes.
- THREADS, MAX_THREADS = *m*—thread-level parallelism on no more than *m* threads.
- ORDERED, NODES, MAX_THREADS = *m*—ordered node-level parallelism on no more than *m* hypernodes.
- ORDERED, THREADS, MAX_THREADS = *m*—ordered thread-level parallelism on no more than *m* threads.
- NODES, CHUNK_SIZE = *n*, MAX_THREADS = *m*—node-level parallelism by chunks of size *n* on no more than *m* hypernodes
- THREADS, CHUNK_SIZE = *n*, MAX_THREADS = *m*—thread-level parallelism by chunks of size *n* on no more than *m* threads.

For more detailed information about optimally parallelizing code on SPP Series machines, consult the *Exemplar Programming Guide.*

On both C Series and SPP Series machines, the CHUNK_SIZE attribute forces the compiler to cyclically execute *n* or fewer iterations until all iterations are executed. For example, the following code uses the PREFER_PARALLEL directive to distribute the loop's work among all threads in chunks of 8 iterations:

```
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 8)
      DO I = 1, 750
         A(I) = C(I) * SIN(B(I))
      END DO
```

By default, the work of the loop is split evenly among the threads in partitions of CHUNK_SIZE iterations. In loops where the chunk size does not evenly divide among the threads, such as

the above example, some threads execute one less iteration. In the example, the loop's 750 iterations are partitioned into 94 chunks; these chunks are distributed among the threads in 92 chunks of 8 iterations and 2 chunks of 7 iterations.

CHUNK_SIZE does not restrict the order in which the loop's iterations start or finish. If a loop's iterations must execute in order because of synchronization requirements, you should use the ORDERED attribute, which also is available on both C Series and SPP Series machines.

The ORDERED attribute forces the compiler to initiate the loop's iterations in order; each iteration will not start until all prior iterations have begun. This is similar to the attribute CHUNK_SIZE = 1, but has the benefit of avoiding possible deadlocks when synchronization depends on the order of the loop's iterations.

```
C$DIR PREFER_PARALLEL(ORDERED)
      DO I = 1, 40
         ...
      END DO
```

ORDERED does not generate synchronization code, so there is no control over the order in which iterations complete. To guarantee proper synchronization of a loop, you may also need to use synchronization intrinsics or critical section directives.

An error occurs when you try to use PREFER_PARALLEL and another parallelizing directive in the same loop nest.

PREFER_PARALLEL_EXT (C Series only)

The PREFER_PARALLEL_EXT directive tells the compiler to parallelize the loop immediately following the directive only if it appears safe to do so. The compiler checks first for actual loop-carried dependencies; if none are found, the loop is parallelized. This directive is effective only in code compiled at optimization level -O3.

PREFER_PARALLEL_EXT is available only on CONVEX C Series machines.

This directive does not prevent interchange of outer loops for vectorization. If you also choose to vectorize this loop, use the PREFER_VECTOR directive. An error occurs when you try to use PREFER_PARALLEL_EXT and another parallelizing directive in the same loop nest.

PREFER_VECTOR (C Series only)

The PREFER_VECTOR directive tells the compiler to vectorize the loop immediately following the directive if it is safe to do so. The compiler checks first for actual recurrences. If no recurrences are found, the compiler tries to interchange the loop to be the innermost loop and vectorize it. This directive is effective only in code compiled at optimization level -O2 or higher.

PREFER_VECTOR is available only on CONVEX C Series machines.

An error occurs when you try to use PREFER_VECTOR and another vectorizing directive in the same loop nest.

PROMOTE_TEST

The PROMOTE_TEST directive allows the compiler to promote tests out of the loop immediately following the directive, replicating code beyond the default conservative limit, but not without bound. This directive is effective only in code compiled at optimization level -O2 or higher. Refer to the *CONVEX Fortran Optimization Guide* Chapter 3, "Vector optimization," for more C Series information.

PROMOTE_TEST_ALL

The PROMOTE_TEST_ALL directive allows the compiler to promote tests out of the loop immediately following the directive, replicating code without bound. This directive is effective only in code compiled at optimization level -O2 or higher. Refer to the *CONVEX Fortran Optimization Guide* Chapter 3, "Vector optimization," for more information.

PSTRIP (C Series only)

The PSTRIP directive tells the compiler that the parallel loop immediately following the directive is to be strip mined using the specified length. This directive is effective only in code compiled at optimization level -O3.

PSTRIP is available only on CONVEX C Series machines.

The format of this directive is

```
C$DIR PSTRIP (integer_constant)
```

where *integer_constant* is an integer constant that specifies the strip-mine length.

Parallel strip mining groups the loop iterations into blocks of $n/(2ep)$, where n is the actual loop trip count and ep is the expected number of processors, which is obtained from the `-ep` option or, in absence of `-ep`, the number of processors in the machine on which the program is running. Each block is executed entirely by a single thread. Parallel strip mining occurs only at `-O3`.

If you do not specify `PSTRIP` directives, the compiler selects a default value appropriate for the architecture of the machine for which you are compiling. The default number of loop iterations to group, or when `-ep` is 1, is 1. At `-O3` when `-ep` is 2 or more, the compiler will use longer strips to reduce the inter-processor overhead.

The `PSTRIP` directive overrides the compiler default and specifies the number of iterations per block to perform. `PSTRIP` cannot be used with vector loops.

Table 34 shows the maximum strip-mine lengths used with the `-O3` and `-ep` options.

Table 34 Maximum parallel strip-mine lengths at `-O3`

Processors (`-ep`)	Default compiler length	`PSTRIP`(*k*) length
1	1	1
More than 1	$\max(n/(2ep),1)$	k

ROW_WISE

Fortran stores arrays in column-major order. Reversing the order of subscripts so that the array is accessed through contiguous rather than noncontiguous memory can improve the efficiency of memory accesses. The `ROW_WISE` directive tells the compiler that the designated arrays have their dimensions reversed. Thus, array elements are stored in a manner consistent with programming languages such as C and Ada. The `ROW_WISE` directive is effective at all optimization levels, including `-no` (no optimization). The format of this directive is

```
C$DIR ROW_WISE (array-name [,array-name...] )
```

The following cautions apply to the use of the `ROW_WISE` directive:

- Implicit array I/O, such as READ(5,*)A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears reversed when viewed in the debugger.
- If the ROW_WISE directive is applied to a dummy argument, the actual argument must also appear in a ROW_WISE directive within the caller. The compiler cannot detect this situation.

The following example illustrates a situation in which use of the ROW_WISE directive can improve performance of a program.

```
DIMENSION A(4,1000)
DO I = 1,4
  DO J = 1,1000
    A(I,J) = 0
  ENDDO
ENDDO
```

Although the preceding example vectorizes, performance is slowed because the array is being accessed with noncontiguous memory (Fortran stores arrays in column-major order). If, however, the code segment in the preceding example is preceded by the directive C$DIR ROW_WISE (A), it is interpreted by the compiler as follows:

```
C$DIR ROW_WISE (A)
DIMENSION A(1000,4)
DO I = 1,4
  DO J = 1,1000
    A(J,I) = 0
  ENDDO
ENDDO
```

The array is now being accessed from contiguous memory, thus increasing the execution speed.

SAVE_LAST (SPP Series only)

The SAVE_LAST directive specifies that all LOOP_PRIVATE variables in the immediately following loop have their values from the final iteration saved for use after the loop's termination. If this directive is not used, then the values of any privatized arrays or variables are indeterminate once the loop terminates.

This directive can be relied upon only for variables that are unconditionally assigned on the final loop iteration. Variables that are conditionally assigned in a loop should not be made

private; instead they should be assigned within an ordered section.

SCALAR

The SCALAR directive prevents the DO loop that follows from being vectorized or parallelized. The body of the loop can still be vectorized or parallelized if an outer loop is interchanged with the SCALAR loop. This directive is effective at all optimization levels.

The SCALAR directive is useful when the iteration count of the loop is too low for the overhead involved in setting up vectorization, or when the numerical results must be the same as for a scalar loop. This directive can also be used to prevent loop interchange, which may not choose the best loop to interchange when the compiler cannot determine the iteration counts of the loops involved. It also can be used to prevent parallel execution of loops when loop overhead is too great for parallel execution.

The results of a vectorized loop can differ from its scalar equivalent. For example, floating-point sum-and-product reduction operators can give different answers because the underlying hardware does not process the operands in sequential order.

```
C$DIR  SCALAR
       DO 10 I = 1,N             !(where N = 2)
       DO 10 J = 1,M             !(where M = 1000)
10     A(I,J) = B(I,J) + C(I,J)
```

In this example, the compiler normally interchanges the I loop with the J loop so that elements of A, B, and C are accessed contiguously. The SCALAR directive ensures that the loop of greater iteration count is retained as the innermost loop.

```
C$DIR   SCALAR
        DO 10 I = 1,N          ! (where N = 2)
C$DIR   SCALAR
        DO 10 J = 1,M          ! (where M = 2)
10      A(I,J) = B(I,J) + C(I,J)
```

In this example, neither iteration count is sufficient to warrant vectorizing the loops.

SELECT (C Series only)

The SELECT directive causes the compiler to generate multiple versions of a loop and to select, at runtime, which version to

execute based on specified trip counts. The compiler generates up to four versions of a loop: scalar, vector, parallel, and parallel-vector. This directive is effective only in code compiled at optimization level -O2 or higher.

SELECT is available only on CONVEX C Series machines.

The format of this directive is

```
C$DIR SELECT (vtrip, ptrip, pvtrip )
```

The arguments *vtrip*, *ptrip*, and *pvtrip* specify the trip (iteration) count at which the compiler is to select vector, parallel, or parallel-vector execution, respectively, for the loop. Parallel-vector execution implies that the loop is vectorized and the strip-mine loop is parallelized.

If you omit a trip count by using two adjacent commas, the compiler selects a default value. If you use an asterisk (*) in place of a trip count, the compiler does not generate code for the corresponding mode. If a specified mode is not available for the loop, the compiler selects a default mode.

If the actual trip count is less than the smallest trip count specified in the directive, the loop runs scalar. If the actual trip count is greater than the largest trip count specified in the directive, the loop runs in the mode of the largest trip count.

Examples:

```
C$DIR SELECT(10,5,20)
C     Run scalar if actual trip count = 1-4.
C     Run parallel if trip count = 5-9.
C     Run vector if trip count = 10-19.
C     Run parallel-vector if trip count ≥ 20.

C$DIR SELECT (0,*,*)
C     Run scalar if loop has no vectorizable code.

C$DIR SELECT (*,*,*)
C     Equivalent to C$DIR SCALAR.
```

SYNCH_PARALLEL (C Series only)

The SYNCH_PARALLEL directive tells the compiler that the following loop is to be executed in parallel; however, instead of ignoring dependencies, the compiler inserts synchronization code that causes the dependencies to be honored at runtime. This directive is effective only in code compiled at the -O3 optimization level.

SYNCH_PARALLEL is available only on CONVEX C Series machines.

Without specific directives, the compiler vectorizes any dependency-free part of the loop; this normally produces superior results. However, if a loop contains much code that is conditionally executed, it may be preferable to parallelize the loop with the SYNCH_PARALLEL directive, particularly if all the dependencies are in seldom-executed branches.

Example:

```
C$DIR SYNCH_PARALLEL
      DO I = 1,32
        IF (A(I).LT.0) THEN
          A(I) = A(I-1) + B(I)
          D(I) = E(I)*F(I)
        ENDIF
      ENDDO
```

This loop might run faster in a machine with four processors than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

TASK_PRIVATE

The TASK_PRIVATE directive declares a list of variables and/or arrays private to the immediately following task. A task is a sequence of code that can be executed in parallel with other tasks. In CONVEX Fortran, tasks are defined using the BEGIN_TASKS, NEXT_TASK, and END_TASKS directives. The TASK_PRIVATE directive is effective only in code compiled at optimization level -O3.

The TASK_PRIVATE directive must immediately precede or appear on the same line as its corresponding BEGIN_TASKS directive. The compiler assumes that variables declared TASK_PRIVATE have no dependencies between the following tasks; therefore no starting or ending value can be assumed for the task-private variable within a task.

The TASK_PRIVATE directive has the form

C$DIR TASK_PRIVATE(*varlist*)

where

varlist

is a list of variables or arrays, separated by commas, that are to be private to each following task. The following tasks are defined by a BEGIN_TASKS directive and one or more NEXT_TASK directives. The scope of the task-private variables is terminated along with the task list when an END_TASKS directive is encountered.

Only variables and statically-sized arrays can be declared private. Dynamic, allocatable, and automatic arrays are not allowed. Structures are not allowed. Including induction variables (i.e. DO loop indices) in *varlist* will yield wrong answers.

If a variable that appears in *varlist* is referenced within a task, it must have been assigned a value previously within that task. Values assigned outside the task list or in other tasks will not be available.

If the variable is referenced after the task list, it must have been assigned a value after the task list. Values assigned inside or before the task list are not available.

For more information about tasks in CONVEX Fortran, see the "BEGIN_TASK" section of this chapter.

THREAD_PRIVATE (SPP Series only)

The THREAD_PRIVATE directive forces one or more specified variables to be stored in thread-private memory.

The THREAD_PRIVATE directive is available only on CONVEX SPP Series machines.

This directive uses the following format:

```
C$DIR THREAD_PRIVATE (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names. Variables listed in *namelist* are stored in thread-private memory rather than shared memory (the default). For more information about memory types available on CONVEX SPP Series machines, refer to the *Exemplar Programming Guide.*

THREAD_PRIVATE_POINTER (SPP Series only)

The THREAD_PRIVATE_POINTER directive places a hidden, compiler-generated pointer to the specified variable in thread-private memory, regardless of the memory class to which

the variable is allocated. This directive is available only on SPP Series machines; it has the form

```
C$DIR THREAD_PRIVATE_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

Additional directives, NODE_PRIVATE_POINTER, NEAR_SHARED_POINTER, and FAR_SHARED_POINTER, place pointers in the other classes of memory.

For detailed information about writing programs that optimally use the memory classes available on CONVEX SPP Series machines, consult the *Exemplar Programming Guide*.

UNROLL

The UNROLL directive reduces loop overhead by replicating the body of the loop that follows. Unrolling is performed only on scalar loops. This directive is effective only in code compiled at optimization level -O2 or higher. It has the form

```
C$DIR UNROLL[(UNROLL_FACTOR=n)]
```

where, if the UNROLL_FACTOR argument is included, its value *n* specifies the number of times to replicate the body of the loop.

To be eligible for unrolling, a loop must have an iteration count that the compiler determines. The compiler unrolls a loop completely only if its iteration count is less than five; otherwise, partial unrolling is performed. On C Series machines, complete unrolling occurs before vectorization, and partial unrolling after vectorization.

UNROLL_AND_JAM

The UNROLL_AND_JAM directive enables the loop unroll and jam transformation for the immediately following loop. The goal is to exploit the use of registers and thus decrease the number of slower main memory accesses. This directive is effective at optimization levels -O2 and -O3. By default on SPP Series machines, the unroll and jam transformation is enabled.

The UNROLL_AND_JAM directive takes the following form:

```
C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=n)]
```

where, if the UNROLL_FACTOR argument is included, its value *n* specifies the number of times to replicate the body of the loop.

See also this chapter's section on the NO_UNROLL_AND_JAM directive and the -uj, -nuj, and -ujn compiler options in Chapter 1. For more information about the unroll and jam transformation on SPP Series machines, refer to the *Exemplar Programming Guide.*

VSTRIP (C Series only)

The VSTRIP directive tells the compiler that the vector loop immediately following the directive is to be strip mined using the specified length. It is especially useful for automatically parallelized vector loops, for example, loops that are vectorized and run with the outer strip parallel. This directive is effective only in code compiled at optimization level -O2 or higher.

VSTRIP is available only on CONVEX C Series machines.

This directive has the following format:

```
C$DIR VSTRIP (integer-constant)
```

where *integer-constant* is an integer constant that specifies the strip-mine length, which must be less than or equal to 128.

Vector strip mining executes the loop in strips of 128 elements by default, and the parallel outer loop runs iterations of the vector loop in parallel.

If you do not specify VSTRIP directives and the compiler doesn't know the number of iterations (or knows that it is larger than 128*ep, where ep is the estimated number of processors), the compiler selects a default value of 128 for the strip-mine length. Also, loops are executed in 128-element strips at optimization level -O2 or if the value of the -ep option is 1. At optimization level -O3 when -ep is 2 or more, the compiler uses more and shorter strips if doing so reduces the length of the longest strip.

The VSTRIP directive overrides the compiler default and specifies a shorter strip-mine length. The shorter strip creates more iterations of the strip-mine loop so that it can be effectively parallelized.

Table 35 shows the maximum strip-mine lengths used with the -O3 and -ep options.

Table 35 Maximum vector strip-mine lengths at `-O3`

Processors (-ep)	Default compiler length	`VSTRIP`(k) length
1	1	128
More than 1	max(min((n+ep-1)/ep,128),8)	k

The actual strip length per iteration is the smaller of the number of iterations remaining to be processed or the maximum length of a strip from the table (either the default or from the directive).

Table 36 shows the maximum and actual vector strip lengths when the system includes four processors (`-ep=4`).

Table 36 Four processor system strip lengths

Trip count	Maximum strip length	Actual strip length(s)
2	8	2
514	128	128, 128, 128, 128, 2 (for the 5 iterations)

Fortran data representations

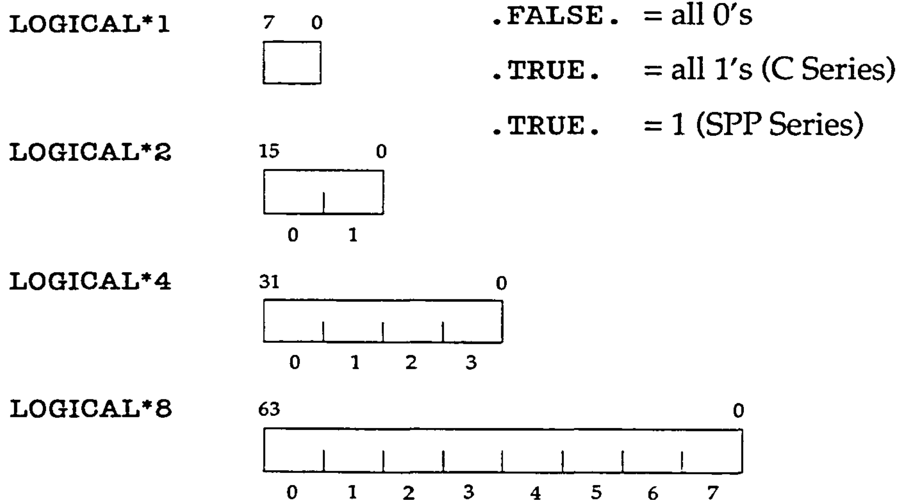
B

This appendix describes the data types supported by CONVEX Fortran and shows how each is stored in memory. The numbers on top of the illustrations are the bit ordering; the numbers on the bottom are the byte ordering.

Logical representation

The leftmost byte (8 bits) is always stored in memory at the lowest byte address. See Figure 20. Note that on SPP Series machines, `.TRUE.` has a value of 1; for example, a `LOGICAL*1` value of `.TRUE.` is represented as 00000001.

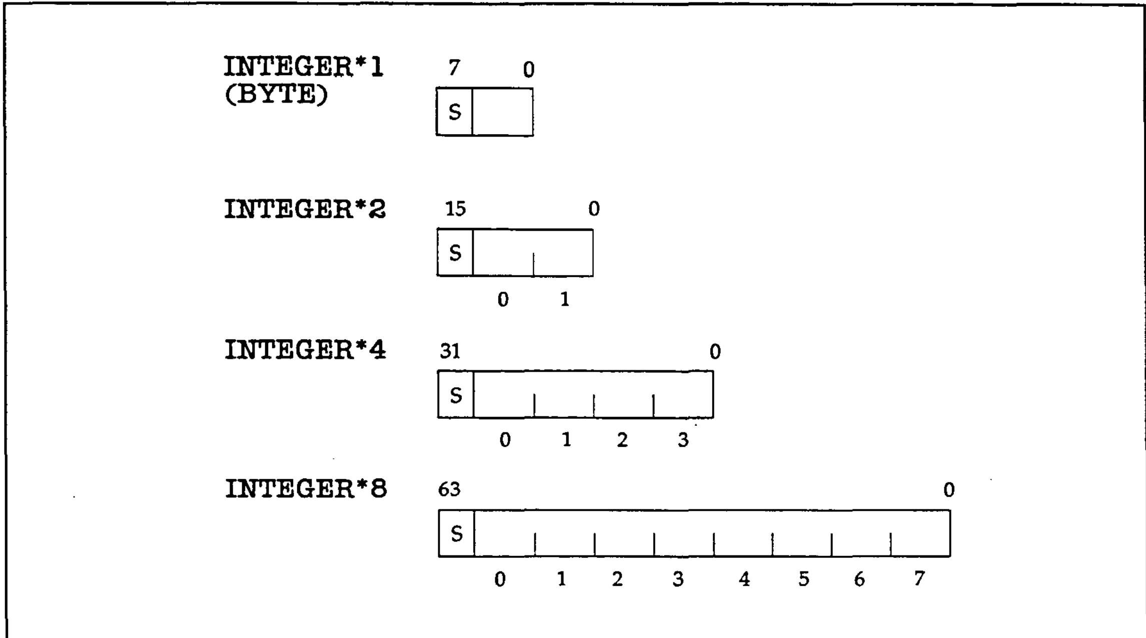
Figure 20 `LOGICAL` data type representation



Integer representation

Integer data is declared with the INTEGER*1 (BYTE), INTEGER*2, INTEGER*4, and INTEGER*8 keywords. In the internal representations, the sign bit (S) is 0 for positive integers and 1 for negative integers, as shown in Figure 21. INTEGER data types use the two's complement format.

Figure 21 INTEGER data type representation



INTEGER*1 values are in the range -128 to +127. INTEGER*2 values are in the range -32,768 to +32,767. INTEGER*4 values are in the range -2,147,483,648 to +2,147,483,647. INTEGER*8 values are in the range -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $+2^{63}-1$).

Real data representation

On both CONVEX C Series and CONVEX SPP Series computers, floating-point variables are declared as REAL*4 (32 bits), REAL*8 (64 bits), or REAL*16 (128 bits).

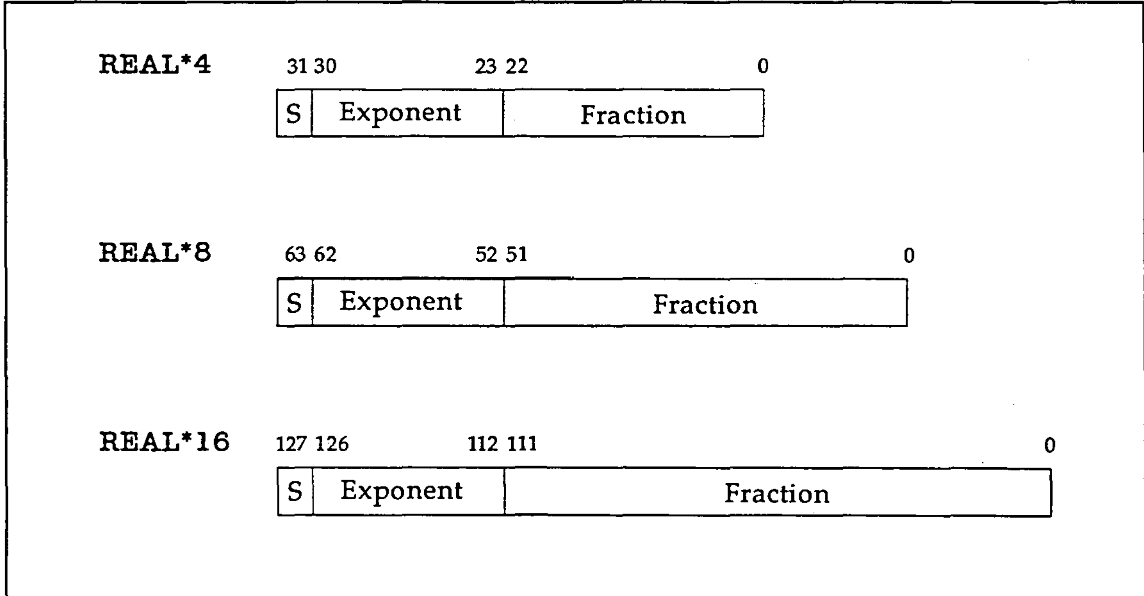
CONVEX SPP Series machines use IEEE representation. On C Series machines, REAL*4 and REAL*8 floating-point data can be represented either in native format or in IEEE format. REAL*16 data cannot be represented in IEEE format on C Series machines.

Note

The CONVEX hardware and runtime libraries support only the processing of data encoded in IEEE format and do not fully conform to the IEEE 754 specifications for arithmetic.

Figure 22 shows the internal representations of floating-point data. The positioning of the sign, exponent, and mantissa (fraction) apply to native and IEEE formats.

Figure 22 REAL data type representation



The range of REAL*16 data is

$$8.405257857780233765656694543304390 \times 10^{-4933}$$

through

$$5.948657476786158825428796633140034 \times 10^{+4931}$$

The ranges for REAL*4 and REAL*8 data are described in the following sections.

Native floating-point (C Series only)

The CONVEX native floating-point representation defines the types of operands shown in Table 37. On C Series machines, code compiled with the -fn option uses CONVEX native floating-point representation; if you do not specify a

Data representations

floating-point format during compilation, your site's default format is used (the site default varies from site to site).

Table 37 Operands defined by CONVEX native floating-point

Operand	Explanation
Normalized	The exponent is not all zeros.
Reserved operand (Rop)	The exponent is 0, the sign is 1, and the fraction can have any value.
Zero	The exponent is 0, the sign is 0, and the fraction can have any value. True zero has a sign of 0, an exponent of 0, and a fraction of 0.

The range of numbers that can be represented in single-precision native floating point is

2.938740×10^{-39}

through

$1.70141 \times 10^{+38}$

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 128; that is, 128 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the left of the implicit 1 bit.

The range of numbers that can be represented in double-precision native floating point is

$5.56268464626801 \times 10^{-309}$

through

$8.98846567431157 \times 10^{+307}$

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1024; that is, 1024 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to

the left of bit position 51. The binary point is to the left of the implicit 1 bit.

IEEE floating-point

The CONVEX IEEE floating-point representation defines the types of operands shown in Table 38. CONVEX SPP Series machines use IEEE represention defined by IEEE/ANSI standard 754-1985. C Series machines provide IEEE representation when the `-fi` compiler option is specified; if you do not specify a floating-point format during compilation, your site's default format is used.

Table 38 Operands defined by CONVEX IEEE floating point

Operand	Explanation
Normalized	The exponent is not all zeros or all ones.
Denormalized	The exponent is 0, the fraction is nonzero, and the sign is 0 or 1. This number is always treated as true zero.
Not a number (NaN)	The exponent is all ones, the fraction is nonzero, and the sign is 0 or 1.
Infinity (Inf)	The exponent is all ones, the fraction is 0, and the sign is 0 or 1.

Data representations

The range of numbers that can be represented in single-precision IEEE floating point is

$$1.1754945 \times 10^{-38}$$

through

$$3.4028232 \times 10^{+38}$$

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the right of the implicit 1 bit.

The range of numbers that can be represented in double-precision IEEE floating point is

$2.225073858507202 \times 10^{-308}$

through

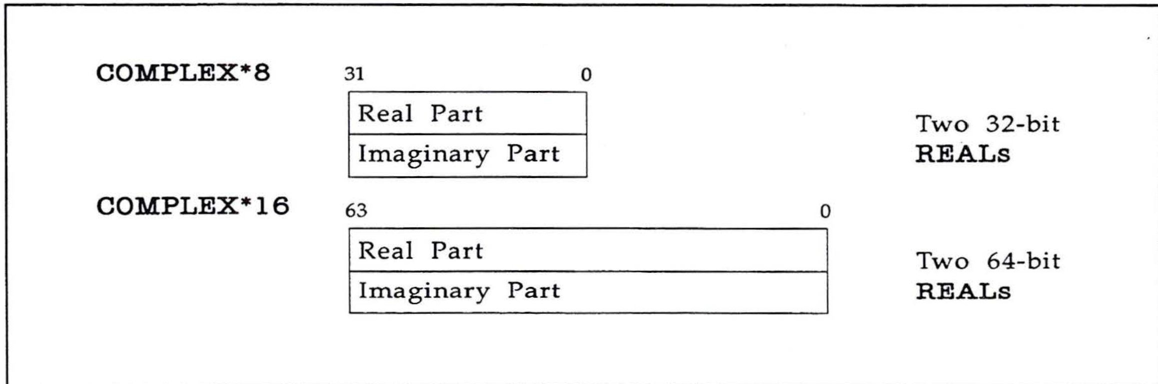
$1.797693134862312 \times 10^{+308}$

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the right of the implicit 1 bit.

Complex representation

Complex numbers are internally represented as shown in Figure 23.

Figure 23 COMPLEX data type representation

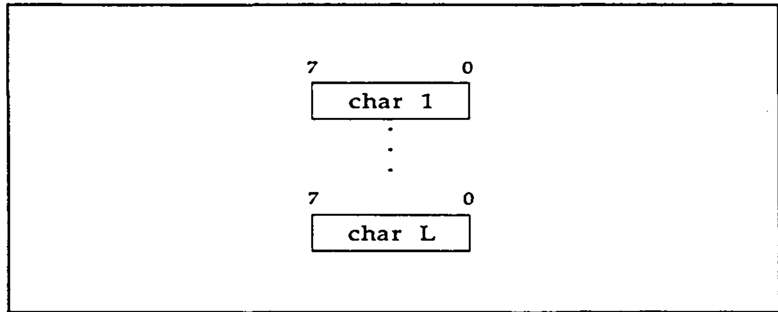


Character representation

A character string is stored internally as a sequence of bytes, as shown in Figure 24. A character constant is limited to 4000

characters. Character strings formed at runtime can be of arbitrary length.

Figure 24 CHARACTER data type representation

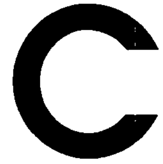


Hollerith representation

A Hollerith constant is stored internally as a sequence of bytes and is limited to 2000 characters.

Data representations

Compiler messages



This chapter describes the messages the CONVEX Fortran compiler reports. There are four types of compiler messages: error, warning, advisory, and vector summarization. Messages that are displayed as a result of an error during a program's execution (runtime error messages) are covered in Appendix D.

When user errors remain after the compiler has completed the syntactic and semantic analyses of a program, the compiler presents an error message and aborts the compilation. An abort also can occur during optimization, for example, as a result of integer truncation during constant folding.

Compiler messages are output to `stderr`, so they appear on your screen unless you redirect them. On C Series machines, `stderr` is unit 0; on SPP Series machines it is unit 7.

Redirecting compiler messages

You can redirect these messages to any specified file using the output redirection characters. To redirect messages for the C-shell only, append the characters `>&` and the file name to the end of the compile command.

For instance,

```
fc file.f
```

sends compiler messages to the screen, while

```
fc file.f >&out
```

when executed under `csh`, sends the messages to the file `out`.

You also can use the `error` utility to insert diagnostic messages into your source file, where they appear as comments. This is a convenient way to locate bugs in your code while editing the source file.

For example,

```
fc foo.f |& error
```

when executed under csh, compiles foo.f and pipes the standard output and standard error output to the error utility, which then inserts the diagnostic messages back in the source file foo.f. See the error(1) man page for details about error.

You can write a simple csh shell script using the error utility to produce listings with embedded error messages that do not modify the source file itself. Refer to the csh(1) man page for details on writing csh scripts.

The -LST compiler option is similar to the error utility and is easier to use. Refer to the "Message and listing options" section of Chapter 1 for more information on -LST.

Compiler error messages

This section lists the compile-time error messages reported by the CONVEX Fortran compiler. The messages listed here are sorted according to message name.

A compiler message includes the line and column number of the text in which the error occurs, the path name of the source file containing the text in error, and a brief description of the error.

Examples:

```
fc: Error on line 7.1 of testprog.f: Label
defined but never referenced.

fc: Warning on line 3.4 of myprog.f: Divide
by zero may occur at runtime.
```

The number before the decimal point is the line number where the error occurs. The number following the decimal point is the column number.

If an internal compiler error occurs, the compiler outputs a message that begins with the words COMPILER ERROR. Such a message should be reported to the CONVEX Technical Assistance Center (TAC).

The -errnames compiler option causes the compiler to print the message name for any compile-time message issued. See Chapter 1, "Compiling programs," for more information on the -errnames option.

arg_equivalenced

fc: *errorclass* on line *n* of *filename*: dummy argument cannot be equivalenced.

An attempt was made to equivalence a dummy argument in the body of a subroutine.

Example:

```
SUBROUTINE SUB(i, j)
EQUIVALENCE (i,k)    ! i and k are
...                  ! dummy arguments
END
```

array_constructor_do_index

fc: *errorclass* on line *n* of *filename*: Implied DO loop indicies in an array constructor must be scalar integer variables.

Implied DO loop indicies in an array constructor must be scalar integer variables.

```
INTEGER A(6)
A = /( (I*2,I=1,6) )/              !OK
A = /( ((I*Z,I=1,2), Z=1,3) )/ !Prohibited
                               ! because Z is REAL
END
```

array_constructor_implied_do_nesting

fc: *errorclass* on line *n* of *filename*: Implied DO loops in an array constructor may not be nested.

Only one level of implied DO loops is currently supported in array constructors

```
INTEGER A(6)
A = /( (I*2,I=1,6) )/              !OK
A = /( ((I*J,I=1,2), J=1,3) )/ !Not allowed
END
```

Compiler messages

array_constructor_type_mismatch

`fc:` *errorclass* `on line` *n* `of` *filename*`: All values in an array constructor must be the same type.`

One of the values in an array constructor was of a different type than another value

Example:

```
INTEGER A(3)
A = /( 1, 2, 3.0 )/    ! mixed integer
                      ! and real types
END
```

array_shape_err

`fc:` *errorclass* `on line` *n* `of` *filename*`: The shapes of` *'name'* `and` *'name'*`, or sections thereof, are not conformable.`

The shapes of arrays in question must be conformable to perform the operation you are attempting.

Example:

```
DIMENSION a(4,5), b(10,3)
...
a = b    ! a and b must be conformable
         ! to do this
END
```

assumed_size_not_parameter

`fc:` *errorclass* `on line` *n* `of` *filename*`:` *name*`, declared as an assumed size array, is not a dummy argument.`

Assumed size arrays can only appear as dummy arguments in subprograms.

Example:

```
PROGRAM foo
REAL a(*)    ! not allowed here
...
END
```

bad_arg_to_row_wise

fc: *errorclass* on line *n* of *filename*: bad argument to ROW_WISE directive.

The argument to ROW_WISE must be an undimensioned array.

Example:

```
C$DIR ROW_WISE(0)    ! 0 is a number
...
END
```

bad_assign_var

fc: *errorclass* on line *n* of *filename*: *name* must be an integer variable to appear in ASSIGN

Only variables of type INTEGER can have labels assigned to them.

Example:

```
ASSIGN 10 to x      ! x is implicitly
                    ! typed REAL
...
END
```

bad_endecode_buffer

fc: *errorclass* on line *n* of *filename*: the third element of the control list of encode or decode must be an array, array element, variable, or character substring.

The third element of the control list of an ENCODE or DECODE statement must be an array, array element, variable, or character substring.

Example:

```
PARAMETER(max = 1000, min = 0, buffer = 10)
...
ENCODE(64, 10, buffer) I,J    ! buffer is
                              ! a PARAMETER
...
END
```

Compiler messages

bad_hex_octet

fc: *errorclass* on line *n* of *filename*: invalid digit in hexadecimal or octal constant

An illegal character was specified in a hex or octal constant.

Examples:

```
   ii = '123fg'X      ! 'g' is not a hex digit
   ii = '1238'O       ! '8' is not an octal
digit
```

bad_iostat_val

fc: *errorclass* on line *n* of *filename*: the value of IOSTAT must be an integer variable or array element.

The IOSTAT keyword had a value which was not an integer variable or integer array element.

bad_kw_val

fc: *errorclass* on line *n* of *filename*: the value of the keyword '*name*' is incorrect.

The value specified for a keyword was not legal. Solution: Change the value to be a legal value for that keyword.

Example:

```
   OPEN (UNIT=3,FORM="grass")  ! grass is
                               ! illegal here
   ...
   END
```

bad_kw_val_type

fc: *errorclass* on line *n* of *filename*: the value of keyword '*name*' is of an incorrect data type.

The value specified for a keyword was not a legal type. Solution: Use the correct type constant of variable for the value.

Example:

```
   REAL unit
   READ(unit,4) a    ! unit wrong type
```

```
...
END
```

cant_be_dimensioned

fc: *errorclass* on line *n* of *filename*: a function or subroutine cannot be dimensioned.

A SUBROUTINE or FUNCTION name cannot be dimensioned.

Example:

```
SUBROUTINE sub
DIMENSION sub(4)      ! can't dimension
                      ! subroutine name
...
END
```

cant_be_in_common

fc: *errorclass* on line *n* of *filename*: *name* is not a variable or array so it may not appear in COMMON.

An entity other than a variable or array appears in a COMMON block.

Example:

```
INTRINSIC LOG
...
COMMON n, a, log      ! log isn't a
                      ! variable or array
...
END
```

cant_be_in_equiv

fc: *errorclass* on line *n* of *filename*: *name* is not a variable or array so it may not appear in EQUIVALENCE.

Only variables and arrays can be equivalenced.

Example:

```
PARAMETER (h = 6.63E-34)
EQUIVALENCE(h, x)     ! h is a PARAMETER
```

Compiler messages

```
...
END
```

cant_convert

fc: *errorclass* on line *n* of *filename*: unable to convert value to required type and precision.

The specified constant value could not be converted to the required type. This is often caused by a constant which is too big, or out of range for the required data type.

Example:

```
INTEGER*1 a
DATA a / 1200 /   ! The value 1200 cannot be
                  ! represented in an INTEGER*1
...
END
```

cant_dim_before_row_wise

fc: *errorclass* on line *n* of *filename*: Cannot dimension *name* before ROW_WISE directive.

Use of the ROW_WISE directive on an array must precede the dimensioning of that array.

Example:

```
      DIMENSION a(10, 400)
C$DIR ROW_WISE(a)    ! must precede DIMENSION
      ...
      END
```

cant_end_do

fc: *errorclass* on line *n* of *filename*: a DO loop cannot terminate with this statement.

A statement label for a DO loop terminating statement appeared on a statement type which cannot be used as the terminating statement of a DO loop. The following statements cannot be used to terminate a DO loop:

DO	block IF
arithmetic IF	ELSE

ELSE IF	END IF
GOTO	assigned GOTO
computed GOTO	RETURN
END	

cant_end_do_warn

`fc:` *errorclass* `on line` *n* `of` *filename*`: a DO loop cannot terminate with this statement.`

You must terminate a DO loop with a labeled executable statement or END DO.

Example:

```
      DO 10 i = 1, 10
      j = i
      IF (j .EQ. 7) THEN
      ...
   10 ENDIF        ! improper terminator for DO
      ...
      END
```

cant_equivalence_across_common_blocks

`fc:` *errorclass* `on line` *n* `of` *filename*`: cannot equivalence` *name* `in common block` *name* `with` *name* `in common block` *name*

A variable in one COMMON block cannot be equivalenced with a variable in another COMMON block.

Example:

```
COMMON /com1/ a /com2/ b
equivalence (a, b)    ! a and b are in
                      ! different COMMON blocks
...
END
```

cant_recover

`fc:  Can't recover from previous errors`

This error indicates the compiler detected an unexpected internal error AFTER encountering a error earlier in the compilation. Removing the earlier error[s] will usually allow the

Compiler messages

compiler to complete successfully or diagnose other errors in the program.

char_star_not_arg

fc: *errorclass* on line *n* of *filename*: *name* is declared CHARACTER*(*) but is not a function, dummy argument or PARAMETER

Only functions, dummy arguments and PARAMETERs can be declared CHARACTER*(*).

Example:

```
CHARACTER*(*) nodummy !illegal declaration
...
END
```

character_convert

fc: *errorclass* on line *n* of *filename*: conversion between '*name*' and CHARACTER is illegal.

An attempt was made to convert a non-CHARACTER variable to CHARACTER or vice-versa. CHARACTER variables can be converted to CHARACTER variables of different lengths according to the rules provided in Chapter 7 of the *Fortran Language Reference,* but conversion between CHARACTER and other data types is not allowed.

Example:

```
CHARACTER*8 name
i = 24
...
name = i    ! illegal conversion
..
END
```

common_dimension

fc: *errorclass* on line *n* of *filename*: COMMON variable '*name*' has a non-constant dimension.

Arrays appearing in COMMON statements must have constant dimensions.

Example:

```
COMMON /badblock/ a(i)
...
END
```

complex_compare

fc: *errorclass* on line *n* of *filename*: the relation for complex operands must be either .EQ. or .NE.

Comparisons other than .EQ. and .NE. are not allowed on COMPLEX operands.

Example:

```
COMPLEX x, z
...
IF(x .LE. z) THEN    ! x and z are
...                  ! type COMPLEX
ENDIF
END
```

concat_with_non_character

fc: *errorclass* on line *n* of *filename*: the operands of concatenation must have type CHARACTER.

An attempt was made to concatenate one or more non-CHARACTER variables.

Example:

```
CHARACTER*8 firstname, lastname
CHARACTER*16 name
INTEGER age
...
firstname = 'Al'
lastname = 'Hofmann'
name = firstname//age  ! age is an INTEGER
...
END
```

directive_form_ignored

`fc:` *errorclass* `on line` *n* `of` *filename*`: Foreign directive ignored.`

This foreign directive is not supported, and is ignored.

do_increment_0

`fc:` *errorclass* `on line` *n* `of` *filename*`: the increment of an implied DO must not be 0.`

A implied DO loop with an increment of zero is not allowed. The loop will never terminate.

Example:

```
INTEGER a(20)
WRITE (6,*) (a(i), i=1,10,0)   ! increment
                               ! cannot be 0
...
END
```

do_variable

`fc:` *errorclass* `on line` *n* `of` *filename*`:` *name* `is not a variable so it cannot be used as a DO loop control variable.`

DO loop control variables must be modifiable.

Example:

```
PARAMETER(n=2)
...
DO n = 1, 10    ! n is a PARAMETER
...
END DO
END
```

do_variable_assigned

`fc:` *errorclass* `on line` *n* `of` *filename*`: the control variable of a DO loop may not be modified in the loop.`

The DO loop index was modified within the DO loop. This is prohibited by the ANSI standard.

Example:

```
DO i=1,10
i=12          ! cannot assign to "i"
...
END DO
...
END
```

do_variable_wrong_type

fc: *errorclass* on line *n* of *filename*: "*name*" is not a real, logical or integer variable and so can not be used as a do loop index.

Do loop indices must be variables of type REAL, LOGICAL, INTEGER, DOUBLE PRECISION, or quad-precision.

Example:

```
CHARACTER*8 badindex
...
DO badindex 1, 10    ! badindex is
...                  ! type CHARACTER*8
END DO
...
END
```

dummy_arg_not_in_expression

fc: *errorclass* on line *n* of *filename*: dummy argument *name* does not appear in the statement function expression.

A dummy argument that appears in the statement function name does not appear in the function definition.

Example:

```
PROGRAM dummy
iavg(i1, i2, i3) = (i1+i2)/3   ! i3 not
                     ! used in definition.
...
END
```

Compiler messages

end_and_rec

fc: *errorclass* on line *n* of *filename*: the END and REC keywords are mutually exclusive.

Both the END and REC keywords were specified. Only one is allowed.

Example:

```
      READ(UNIT=*, REC = 2, END = 10) X
      ...
   10 ...
      ...
      END
```

end_of_program_in_do

fc: *errorclass* on line *n* of *filename*: END found before ENDDO or labeled statement in a DO loop.

An END statement for the subprogram was seen before all DO loops were completed. Either a statement label is missing, an END DO statement is missing, or the END statement was misplaced.

Example:

```
INTEGER  q
k=0
DO q=1,100
   k=k+1
   ...
END            ! no END DO statement
```

end_of_program_in_if

fc: *errorclass* on line *n* of *filename*: END found before ENDIF in IF .. THEN block.

An END statement for the subprogram was seen before all block IF statements were completed. Either an END IF statement is missing, or the END statement was misplaced.

Example:

```
INTEGER  q
IF (q .eq. 0) THEN
   q = -1
```

```
    ...
    END        ! no END IF statement
```

entity_initialized

fc: *errorclass* on line *n* of *filename*: you can only initialize variables and arrays.

Some item other than an array or variable was initialized. Only arrays and variables may be initialized. FUNCTION names, SUBROUTINE names, COMMON block names, NAMELIST names, and so on, cannot be initialized.

Example:

```
    FUNCTION aaa
    INTEGER aaa
    DATA aaa/4/  !can't initialize function aaa
    ...
    END
```

equiv_conflict

fc: *errorclass* on line *n* of *filename*: equivalence conflict: inconsistent equivalence.

A conflict was detected in EQUIVALENCE processing. A variable was specified to exist at two different locations. Redundant EQUIVALENCEs are allowed, but must not require a variable to exist in two different memory locations.

Example:

```
    INTEGER  a(5), b(5), c
    EQUIVALENCE (a(1), b(2)), (b(5),c)
    EQUIVALENCE (a(5),c)
             ! "c" must overlay a(5) and
             ! b(5), but b(5) must overlap
             ! a(4).  "c" cannot overlap both
             ! a(5) and b(5) unless a(5) and
             ! b(5) are overlapped also.
    ...
    END
```

Compiler messages

extending_common

fc: *errorclass* on line *n* of *filename*: extending common block *name* on left via variable *name*

The equivalence you have specified might cause a reference to storage before the start of the common block.

Example:

```
DIMENSION a(5), b(5)
COMMON /com1/ a
...
EQUIVALENCE (a(1),b(2))    ! this puts b(1)
                           ! before start of block
...
END
```

file_and_unit

fc: *errorclass* on line *n* of *filename*: FILE and UNIT specifiers must not both appear.

Both a FILE= and a UNIT= were specified for an INQUIRE statement. Only one is allowed.

fork_with_no_endfork

fc: *errorclass* on line *n* of *filename*: BEGIN_TASKS directive with no END_TASKS

No END_TAKS directive was detected to match a BEGIN_TASKS .

Example:

```
C$DIR BEGIN_TASKS
      PI = 3.14
C$DIR NEXT_TASK
      c = 2.99E8
      h = 6.63E-34
      ...
      END
```

fork_within_fork

fc: *errorclass* on line *n* of *filename*: embedded BEGIN_TASKS directives disallowed

A BEGIN_TASKS directive was encountered before the END_TASKS directive necessary to end a preceding task list.

BEGIN_TASKS directives cannot be nested. A task list must end with an END_TASKS directive before another task list can begin.

Example:

```
 C$DIR BEGIN_TASKS
       pi = 3.14
 C$DIR NEXT_TASK
       c = 3.0E8
 C$DIR BEGIN_TASKS    ! this directive out
                     ! of order
       h = 6.63E-34
       ...
       END
```

func_call_parallel_io

`fc:` *errorclass* `on line` *n* `of` *filename*`: if function` *name* `performs any I/O operation, deadlock could result.`

If the function you are calling in an I/O list attempts to do I/O itself or is parallelized, deadlock could result.

Example:

```
    EXTERNAL foo
    ...
    a = 1.0
    PRINT*, foo(a)     ! foo must not do I/O
                      ! or go parallel
    ...
    END
```

illegal_char_in_label

`fc:` *errorclass* `on line` *n* `of` *filename*`: Illegal character in label field.`

Labels must be decimal integers between 1 and 99999.

Example:

```
    10X    ...    ! 'X' not allowed
           ...
           END
```

Compiler messages

illegal_in_where

fc: *errorclass* on line *n* of *filename*: Only array assignment statements are allowed in a block WHERE construct.

You are attempting to perform an operation other than an array assignment in a WHERE construct.

Example:

```
DIMENSION a(10)
...
WHERE (a .EQ. 0)
PRINT *, "a equals 0"    ! not allowed
                        ! in WHERE
ENDWHERE
...
END
```

illegal_type_length

fc: *errorclass* on line *n* of *filename*: illegal length for type.

The length you specified in a type statement is not valid for that type of variable.

Example:

```
INTEGER*3 x    ! no such thing as INTEGER*3
...
END
```

illegal_value_argument

fc: *errorclass* on line *n* of *filename*: illegal type for argument to %VAL.

Arguments to %VAL must be variables of type INTEGER, REAL or LOGICAL, from 1 to 4 bytes in length.

Example:

```
REAL*8 x
...
CALL foo(%VAL(x))    ! x is type REAL*8
...
END
```

implicit_type_not_available

fc: *errorclass* on line *n* of *filename*: implicit type required for *name* but IMPLICIT NONE has been seen.

All variables must be explicitly typed when the IMPLICIT NONE statement is being used.

Example:

```
    IMPLICIT NONE
    i = 1    ! i undeclared
    ...
    END
```

impossible_do_nesting

fc: *errorclass* on line *n* of *filename*: the DO loop nesting implied by this label is impossible.

DO loops must be completely nested. One DO loop is not allowed to partially overlay the range of another DO loop.

Example:

```
    DO 10 i=1,5
    DO 20 j=1,4     !this must terminate
                    !before line 10
    ...
 10 CONTINUE
    ...
 20 CONTINUE
    ...
    END
```

include_failed

fc: *errorclass* on line *n* of *filename*: Can't find include file '*name*'

The user specified INCLUDE file could not be found. Either it does not exist, the directory it is in was not specified with the -I option, or the permissions on the file or its containing directory are incorrect.

Compiler messages

include_nesting

fc: *errorclass* on line *n* of *filename*: include file nesting is limited to *limit* files; can't include *'name'*

The compiler can only handle concurrently nested INCLUDE files to the specified depth. One possible cause is a "recursive" INCLUDE, where an INCLUDE file includes itself, either directly or indirectly.

include_syntax

fc: *errorclass* on line *n* of *filename*: Syntax error in include statement

The INCLUDE statement was not specified correctly. This is usually caused by an improper file name specification.

Example:

```
INCLUDE myfile      ! the file name must
                    ! be enclosed in quotes
...
END
```

include_syntax_eof

fc: *errorclass* on line *n* of *filename*: End of file in include statement - can't continue.

An INCLUDE statement was not complete before the compiler detected the end of the input file. The file name was probably specified incorrectly or missing.

Example:

```
INCLUDE "myfile     ! no closing quote
...
END
```

incorrect_type

fc: *errorclass* on line *n* of *filename*: an argument to an intrinsic has an incorrect type.

An argument to an intrinsic was of the wrong type for that particular intrinsic.

Example:

```
INTRINSIC LOG
z = LOG (.TRUE.)     ! .TRUE. is wrong type
                     ! argument for LOG
...
END
```

init_uncommon

fc: *errorclass* on line *n* of *filename*: only common can be initialized in a BLOCK DATA subprogram.

Only variables and arrays in a COMMON block can be initialized in a BLOCK DATA subprogram. All other items are local to the BLOCK DATA and are not accessible anywhere else in the program.

Example:

```
BLOCK DATA bd1
COMMON //a,b,c
data a,z/3.0,4.0/    ! "z" is not in COMMON
...
END
```

intrinsic_not_passable

fc: *errorclass* on line *n* of *filename*: this intrinsic cannot be passed as a function actual argument.

A generic INTRINSIC function was passed as an argument to another subprogram. Some generics cannot be passed to a subprogram because the compiler doesn't know which specific intrinsic to pass. Solution: Pass the appropriate specific intrinsic.

Example:

```
INTRINSIC max
CALL sub1(max)    ! can't pass generic max
...
END
```

invalid_array_use

fc: *errorclass* on line *n* of *filename*: an array reference must be subscripted.

An array was referenced without subscripts in a context requiring subscripts.

Example:

```
INTEGER a(44)
CALL subl(a)        ! ok
a = 3            ! not allowed when -nof90
                 ! is specified
...
END
```

invalid_f66external

fc: *errorclass* on line *n* of *filename*: a FORTRAN-66 form of EXTERNAL is illegal in a FORTRAN-77 program.

Using the form of the FORTRAN 66 EXTERNAL statement necessary to specify a user-defined external subprogram whose name is identical to a CONVEX Fortran intrinsic in absence of the -f66 option will cause this error.

Example:

```
EXTERNAL *sqrt    ! this is how an external
             ! user-defined sqrt function is
             ! specified in FORTRAN 66
...
END
```

invalid_type_for_generic

fc: *errorclass* on line *n* of *filename*: this generic intrinsic cannot have the type previously assigned.

The generic intrinsic was declared with an inappropriate type.

Example:

```
LOGICAL LOG
INTRINSIC LOG
...
```

```
x = LOG(y)
...
END
```

invalid_unit_star

fc: *errorclass* on line *n* of *filename*: '*' is not a valid unit specifier in this context.

A star ("*") was specified as a UNIT in a context which does not allow the use of "*".

kw_not_this_stmt

fc: *errorclass* on line *n* of *filename*: the '*name*' keyword is illegal in *name* statements.

A recognized keyword was seen in a statement which does not allow that particular keyword. Solution: remove the offending keyword.

Example:

```
CLOSE (FMT=4)    ! FMT not allowed in CLOSE
...
END
```

kw_val_must_be_label

fc: *errorclass* on line *n* of *filename*: the value of the keyword '*name*' must be a label.

Certain keywords require the value associated with them to be the same as one of the statement labels in the same subprogram. Solution: Make sure the value for the keyword specifies an appropriate statement label.

Example:

```
SUBROUTINE doit
WRITE(6,FMT=5)    ! no statement label 5
...
END
```

Compiler messages

kw_val_must_be_positive

`fc:` *errorclass* `on line` *n* `of` *filename*`: the value of the keyword '`*name*`' must be positive.`

The valued specified for a keyword was not positive, but is required to be. Solution: Correct the offending value.

Example:

```
WRITE (6, FMT=-4)    ! FMT must be positive
...
END
```

labelled_continuation

`fc:` *errorclass* `on line` *n* `of` *filename*`: Label appears on continuation line.`

Labels are allowed in columns 1-5 only; column 6 is reserved for continuation line identifiers.

Example:

```
123456 I=10
      ...
      END
```

lower_exceeds_upper

`fc:` *errorclass* `on line` *n* `of` *filename*`: lower bound of` *name* `exceeds upper bound.`

An array was dimensioned with one dimension where the specified (or implicit) lower bound was greater than the upper bound.

Example:

```
REAL a(3,-3:-2,-5:-6) ! the last
                      ! dimension (-5:-6) is
                      ! erroneous
...
END
```

member_required

fc: *errorclass* on line *n* of *filename*: *'name'* is not a component of variable *'name'*

A record component was specified for a variable which is not declared for that variable.

Example:

```
STRUCTURE /structl/
   INTEGER aaa
END STRUCTURE
RECORD /structl/ zz
zz.aa = 1          !should have been zz.aaa
...
END
```

misplaced_else

fc: *errorclass* on line *n* of *filename*: an ELSE statement cannot appear here in this IF .. THEN block.

An IF...THEN statement has an improperly placed ELSE statement.

Example:

```
IF(i .EQ. 1) THEN
   j = 10
ELSE
   j = 100
ELSE         ! can't have two ELSEs in a row
   j = 1000
ENDIF
...
END
```

misplaced_elseif

fc: *errorclass* on line *n* of *filename*: an ELSEIF statement cannot appear here in this IF .. THEN block.

An IF...ELSE IF statement has an improperly placed ELSE IF statement.

Compiler messages

Example:

```
IF(i .EQ. 1) THEN
   j = 10
ELSE
   j = 100
ELSE IF(i .EQ. 3)    ! doesn't belong here.
   j = 1000
ENDIF
...
END
```

missing_kw

fc: *errorclass* on line *n* of *filename*: *'name'* is not a valid, valueless keyword.

A keyword which requires an additional value was seen without a value. Solution: Add an appropriate value for the keyword, or remove the keyword from the statment.

Example:

```
WRITE (6,*,IOSTAT)
...
END
```

missing_subscripts

fc: *errorclass* on line *n* of *filename*: number of subscripts < dimension.

An array was referenced without the required number of subscripts. Most array references (except in SUBROUTINE and FUNCTION calls) must have the same number of subscripts as dimensions in the array.

Example:

```
INTEGER a(5,4)   ! two dimensional array "a"
a(1,2,3) = 0     ! 3 subscripts for a two
                 ! dimensional array is not
                 ! allowed
...
END
```

more_than7dimensions

fc: *errorclass* on line *n* of *filename*: array *name* has more than 7 dimensions.

FORTRAN arrays are limited to a maximum of 7 dimensions.

Example:

```
INTEGER IARR(10,10,10,10,10,10,10,10)
                         ! IARR has 8 dimensions
...
END
```

multiple_common_def

fc: *errorclass* on line *n* of *filename*: *name* may not appear here because it previously appeared in COMMON.

A variable appeared twice, in different COMMON blocks, or possibly twice in the same COMMON block.

Example:

```
COMMON abc
COMMON /cb/ abc          ! same variable in
                        ! two common blocks
...
END
```

multiple_declarations

fc: *errorclass* on line *n* of *filename*: this variable already has been declared.

A variable was declared more than once.

Example:

```
INTEGER a
INTEGER a    ! redundant declaration
...
END
```

Compiler messages

multiple_dimensions

`fc:` *errorclass* `on line` *n* `of` *filename*`:` *name* `already has been dimensioned.`

An array was dimensioned twice.

Example:

```
DIMENSION a(4)
INTEGER a(5)        ! two attempts to
                    ! dimension the
                    ! same array
...
END
```

multiple_types

`fc:` *errorclass* `on line` *n* `of` *filename*`: this variable already has been assigned a type.`

The specified variable was given two types.

Example:

```
INTEGER a
INTEGER*2 a     ! "a" can't have two types
...
END
```

must_be_array

`fc:` *errorclass* `on line` *n* `of` *filename*`: only arrays can be subscripted.`

Some item other than an array was subscripted. This may also occur when a `FUNCTION` name is called with arguments, but a previous erroneous use of the `FUNCTION` name indicated to the compiler that said name was not a `FUNCTION` name.

Example:

```
SUBROUTINE sub1
sub1(3,4) = 0  ! attempting to assign
               ! into subscripted
               ! subroutine
...
END
```

must_be_character

fc: *errorclass* on line *n* of *filename*: substrings must be initialized to character values.

A CHARACTER variable substring in a DATA statement can only be initialized with character data.

Example:

```
CHARACTER *5 c
data c(1:3)/4/      ! can't initialize
                   ! character variable
                   ! with integer
...
END
```

must_not_be_arg

fc: *errorclass* on line *n* of *filename*: dummy arguments cannot be initialized.

Dummy arguments cannot be initialized.

Example:

```
FUNCTION f1 (a,b)
DATA a/2.71828/
...
END
```

no_arg_to_directive

fc: *errorclass* on line *n* of *filename*: ROW_WISE directive must have arguments.

The ROW_WISE directive must include one or more arguments.

Example:

```
C$DIR ROW_WISE
      ...
      END
```

Compiler messages

no_do_block

fc: *errorclass* on line *n* of *filename*: no matching DO statement for this ENDDO.

An END DO statement without a matching DO statement was seen. This may be caused by an extra END DO statement, or by a syntax error in the DO statement which causes that DO statement to be treated as an assignment statement.

Example:

```
DO I=1.3            ! This is an assignment
   ...              ! statement, since there
   ...              ! was no comma
END DO
...
END
```

no_if_block

fc: *errorclass* on line *n* of *filename*: no matching IF .. THEN statement for this statement.

An END IF statement without a matching block IF statement was seen. This may be caused by a missing IF statement, an extra END IF statement, or a syntax error in the IF statement which causes that IF statement to be treated as some other type of statement.

Example:

```
IF (i) =HEN    ! possible statement function
   ...         ! but not an IF statement
END IF
...
END
```

no_nondo_vars

fc: *errorclass* on line *n* of *filename*: variables (except implied DO variables) may not be used in subscripts here.

Only implied DO loop indices may be used as a subscript in certain subscript calculations.

Example:

```
REAL z(5)
DATA z(j),i=1,4 / 4*1.0 /
                ! "j" is not an implied DO loop
                ! index in this statement.
...
END
```

no_recursion

fc: *errorclass* on line *n* of *filename*: recursive calls are not permitted in FORTRAN

Recursive calls are not permitted in Fortran without using the -re option. Solution: Don't call the subprogram recursively, or compile the routine with the -re option.

Example:

```
SUBROUTINE s
...
CALL s      ! subroutine recursively
            ! calls itself
...
END
```

no_unit_specifier

fc: *errorclass* on line *n* of *filename*: a unit must be specified, but no unit specifier was found.

Certain I/O statements require a UNIT to be specified. Solution: Add a UNIT number to the statement.

Example:

```
WRITE(FMT=2)    ! UNIT required for WRITE
...
END
```

non_constant_complex_component

fc: *errorclass* on line *n* of *filename*: *name* appears in a complex constant but is not a PARAMETER identifier.

COMPLEX constants must consist of real constants and/or real parameters.

Example:

```
COMPLEX i
i = (1.5D0, A)    ! A is a variable
...
END
```

non_logical_if_expression

`fc:` *errorclass* `on line` *n* `of` *filename*`: The expression in a logical if statement must be logical or integer.`

The expression in a logical `IF` statement must evaluate to a value of type `LOGICAL` or type `INTEGER`.

Example:

```
REAL x, y
...
IF (x-y) THEN    ! x and y are REALs
...
END IF
END
```

not_intrinsic

`fc:` *errorclass* `on line` *n* `of` *filename*`: this identifier is not a valid intrinsic name.`

The name specified on an `INTRINSIC` statement was not recognized as an `INTRINSIC`. Either it was misspelled or it is not a valid intrinsic in the current compiler mode. Certain Cray and Vax intrinsics are only recognized in the corresponding compiler mode.

Example:

```
INTRINSIC foo    ! foo not an intrinsic
...
END
```

not_subroutine

fc: *errorclass* on line *n* of *filename*: a non-subroutine cannot be called as a subroutine.

A variable which is not a SUBROUTINE cannot be used in a CALL statement (as the subroutine name).

Example:

```
integer a
call a(4)   ! "a" was typed INTEGER, so it
            ! cannot also be a SUBROUTINE
...
END
```

parameter_assignment

fc: *errorclass* on line *n* of *filename*: a PARAMETER cannot be assigned a value.

An attempt was made to assign a value into an identifier that was previously defined in a PARAMETER statement.

Example:

```
PARAMETER (x = 2)
...
x = 3    ! x is a PARAMETER
...
END
```

parameter_misuse

fc: *errorclass* on line *n* of *filename*: a parameter cannot be subscripted.

Programmer named constants (PARAMETERs) cannot be subscripted.

Example:

```
PARAMETER (pi=3.141592654)
A = pi(2)    ! parameter pi can't
             ! be subscripted
...
END
```

Compiler messages

program_name

fc: *errorclass* on line *n* of *filename*: the PROGRAM name cannot be used this way.

The PROGRAM name cannot be used as a name for an external procedure, block data subprogram, common block, variable, array or constant in the same program.

Example:

```
PROGRAM progname
...
progname = 1  ! progname can't be used here
...
END
```

program_too_complex

```
fc:
 >>>>>   C O M P I L E R   E R R O R   <<<<<
 >>>>> See your system manager for help <<<<<
 This program is too complex (name).
```

The compiler has exceeded some internal limit. Please contact your system administrator and submit this program to CONVEX.

recursive_statement_function

fc: *errorclass* on line *n* of *filename*: a statement function cannot be used in its defining expression.

A statement function cannot be defined in terms of itself.

Example:

```
recfun(x) = x*recfun(x)   ! recfun is
                    ! recursively defined
...
END
```

ref_type

fc: *errorclass* on line *n* of *filename*: Illegal type for argument to %REF

Arguments to %REF must not be Hollerith constants.

Example:

```
CALL foo(%REF(3Hbud))    ! Hollerith
      ! constant 3Hbud not allowed here
...
END
```

repeated_kw

fc: *errorclass* on line *n* of *filename*: the keyword '*name*' has been repeated.

A keyword appeared twice in the statement. Solution: Remove one of the uses of the keyword.

Example:

```
READ (UNIT=3,FMT=9,UNIT=2)    ! UNIT
                             ! appears twice
...
END
```

repetition

fc: *errorclass* on line *n* of *filename*: repetition factors must be unsigned integer constants.

A repetition count in a data initializer must be an unsigned integer constant.

Example:

```
REAL a(4)
DATA a / 4.0*99.0/      ! 4.0 is an invalid
                       ! repetition
...
END
```

return_not_in_subprogram

fc: *errorclass* on line *n* of *filename*: RETURN statements may only appear in subprograms.

RETURN statements can only appear in subprograms.

Example:

```
PROGRAM foo
...
```

Compiler messages

```
RETURN   ! cannot appear in a main program
END
```

row_wise_must_be_array

fc: *errorclass*: *name* not multi-dimensional scalar array or is adjustable in ROW_WISE directive.

An argument to ROW_WISE cannot be declared as a scalar variable or dimensioned as an adjustable array.

Example:

```
      INTEGER a
C$DIR ROW_WISE(a)    ! a is not an array
      ...
      END
```

section_not_allowed

fc: *errorclass* on line *n* of *filename*: The array object or section '*name*' is not allowed in this context.

The array object or section in question cannot be used in the way in which you are attempting to use it.

Example:

```
dimension a(4,5)
...
print*, a(1:3, 2:4)    ! illegal use of
                       ! array section
...
END
```

short_equiv

fc: *errorclass* on line *n* of *filename*: equivalence list must contain at least two names.

An EQUIVALENCE list must contain at least two names.

Example:

```
EQUIVALENCE (a,b,c),(g)    ! "g" is not
                ! EQUIVALENCED to anything
...
END
```

sort_failed

```
fc:
 >>>>>   C O M P I L E R   E R R O R   <<<<<
 >>>>> See your system manager for help <<<<<
 Can't sort error file for listing (system
 command=name).
```

When the compiler tried to call the system sort routine (usually /bin/sort) an error occurred. If this fails repeatedly, contact your system administrator.

subroutine_as_function

fc: *errorclass* on line *n* of *filename*: a subroutine cannot be called as a function.

A SUBROUTINE cannot be referenced as if it was also a function.

Example:

```
CALL z(1)
var = z(33) !a SUBROUTINE cannot return a
            !value as if it were a FUNCTION
...
END
```

subscript_count

fc: *errorclass* on line *n* of *filename*: wrong number of subscripts for this array.

The wrong number of subscripts was specified for an array reference.

Example:

```
REAL a(3,3)
a(1,2,3) = 1
...
END
```

subscript_range

fc: *errorclass* on line *n* of *filename*: subscript expression is out of bounds.

An array subscript was specified which is not in the declared range for the corresponding dimension. This practice is discouraged, even if the computed array address is still within

Compiler messages

the bounds of the actual array. This practice will often lead to unexpected results when optimization levels above "-no" are used.

Example:

```
INTEGER a(3,4)
a(4,3) = 0          ! the 1st subscript is
                    ! out of range
...
END
```

substr_first_exceeds_second

`fc:` *errorclass* `on line` *n* `of` *filename*`: the first substring index exceeds the second`

The first substring index must be less than or equal to the second.

Example:

```
CHARACTER*16 whole, wrong
wrong = whole(6:5)              ! 6>5
...
END
```

substr_index_exceeds_length

`fc:` *errorclass* `on line` *n* `of` *filename*`: index number` *limit* `of substring exceeds string length`

Character substring indices must not exceed the length of the string.

Example:

```
CHARACTER*16 whole, toomuch
toomuch = whole(1:18)  ! whole is only
                       ! 16 characters long
...
END
```

substr_index_must_be_positive

`fc:` *errorclass* `on line` *n* `of` *filename*`: index number` *limit* `of substring is less than 1`

Character substring indices must be positive.

Example:

```
CHARACTER*16 whole, half
half = whole(0:8)    ! 0 illegal as
                     ! substring index
...
END
```

symbol_type_redeclared

fc: *errorclass* on line *n* of *filename*: the type of *name* has already been declared.

Variables can only be declared once.

Example:

```
INTEGER x
REAL x    ! x already declared type INTEGER
...
END
```

syntax_error_at_end_statement

fc: *errorclass* on line *n* of *filename*: Syntax error: premature end of statement.

This error is issued when the complier recognizes a statement that is incomplete.

too_few_values

fc: *errorclass* on line *n* of *filename*: too few data values specified.

A DATA statement contained more variables and array locations to be initialized than initial values.

Example:

```
REAL z(3)
DATA a,b,c / 2.0, 3.0 / ! no value for "c"
DATA z / 1.0, 4.0 /    ! no value for "z(3)"
...
END
```

too_many_subscripts

`fc:` *errorclass* `on line` *n* `of` *filename*`: too many subscripts specified.`

An array was referenced in a DATA or EQUIVALENCE statement with more subscripts than the array was dimensioned with.

Example:

```
INTEGER a(5,5)
EQUIVALENCE (b, a(1,2,3))
...
END
```

too_many_values

`fc:` *errorclass* `on line` *n* `of` *filename*`: too many data values specified.`

A DATA statement contained more data values than variable and array locations to be initialized.

Example:

```
REAL a(3)
DATA a/4*55.0/    ! 4 data values for
                  ! 3 locations
...
END
```

two_main_programs

`fc:` *errorclass* `on line` *n* `of` *filename*`: This source file contains two main programs`

Two or more PROGRAM statements appear in the source file.

Example:

```
PROGRAM firstprog
...
END
PROGRAM secondprog    ! only one PROGRAM
                      ! statement allowed
...
END
```

unit_must_be_integer

fc: *errorclass* on line *n* of *filename*: the unit specifier must be an integer.

A unit specifier was seen which was not of type integer. Solution: Only use INTEGER constants and variables for UNIT specifiers.

Example:

```
REAL x
READ(UNIT=x,FMT=2)    ! x is wrong type for
                      ! UNIT specifier
...
END
```

unmatched_commons

fc: *errorclass* on line *n* of *filename*: Unmatched type or offset of common entities *name* and *name* prevents inline substitution.

Common variables must be of the same type and offset for inlining to work.

Example:

```
PROGRAM inlinemain
COMMON /com1/ i,j
...
t = a(4)
...
END

INTEGER FUNCTION a(k)
COMMON /com1/ x,y     ! x,y not same
                      ! type as i,j above
...
a = x + k
RETURN
END
```

Compiler messages

unsupported_open_kw

fc: *errorclass* on line *n* of *filename*: '*name*' keyword in OPEN not supported on this architecture - ignored

The specified keyword is not supported in the OPEN statement on this architecture.

var_unassigned

fc: *errorclass* on line *n* of *filename*: variable '*name*' not defined by assign statement.

Before an INTEGER variable can be used as a statement label in a GOTO statement it must be given a statement label value in an ASSIGN statement. Arithmetically assigned values cannot be used in place of statement labels.

Example:

```
      m = 10   ! need to use ASSIGN statement
      GOTO m
   10 ...
      END
```

variable_subscripted

fc: *errorclass* on line *n* of *filename*: a non-dimensioned variable cannot be subscripted.

A variable cannot be subscripted. This error may occur if a function name is referenced without any arguments, which tells the compiler it must be a simple variable instead of a function. Later references to the function name will cause this error to be generated.

Example:

```
INTEGER a
b = a
c = a(44)
...
END
```

warn_contained_loop_exit

fc: *errorclass* on line *n* of *filename*: *name* directive ignored - loop or inner loop has exit

The directive you specified cannot be used on a loop with an exit.

Example:

```
C$DIR FORCE_PARALLEL
      DO i = 1, 1000
         ...
         IF(a(i) .EQ. 1) GOTO 10    ! loop exit
         ...
      ENDDO
10    ...
      END
```

warn_setjmp_wont_work

fc: *errorclass* on line *n* of *filename*: *WARNING* use of the predefined routine '*name*' may cause incorrect results when optimizing: if this is not a user-written routine, optimization must be disabled by using the '-no' compiler switch; use of a user-written routine with this name will work properly at any optimization level.

The specified routine, if predefined, can cause incorrect results if optimized. Either compile at -no or use a user-written routine instead.

Example:

```
   i = SETJMP(IENV)
   ...
   END
```

wrong_num_actual_args

fc: *errorclass* on line *n* of *filename*: wrong number of arguments for this statement function.

A statement function was invoked with a different number of arguments than it was declared with.

Example:

```
sf(a,b,c) = (a + b*c) ** 2
z = sf(2,3)    ! sf needs 3 arguments
...
END
```

wrong_num_args

`fc:` *errorclass* `on line` *n* `of` *filename*`: an intrinsic has been used with an incorrect number of args.`

An intrinsic was called with the wrong number of arguments.

Example:

```
z = SQRT (1.0,2.0,3.0)
...
END
```

zero_length_bit_pattern

`fc:` *errorclass* `on line` *n* `of` *filename*`: Illegal zero length bit string` *name*

A hex or octal constant contains no digits.

Example:

```
i = ''X    ! i is empty hex constant
...
END
```

zero_length_hollerith

`fc:` *errorclass* `on line` *n* `of` *filename*`: a zero length Hollerith constant is illegal.`

A Hollerith constant of length zero was encountered. This is not allowed. Solution: use "1h " if appropriate, or possibly a character constant (i.e. " ").

Example:

```
DATA k /0h/    ! illegal Hollerith constant
...
END
```

Compiler messages

Runtime error messages

D

The runtime library reports errors that are encountered during a program's execution. Runtime errors fall into three categories: system-detected errors, arithmetic errors, and I/O errors.

The runtime library provides default error processing and error message generation. On both CONVEX C Series and SPP Series machines, all error messages are written to `stderr`. On C Series machines, `stderr` is unit 0; on SPP Series machines it is unit 7.

System-detected errors

System-detected errors can be returned either by the Fortran I/O library or by the Fortran utility library. When the I/O library returns a system error, the error is in the form of an I/O error message. The utility library returns system errors in the form of error numbers, which are returned by utility functions (as described in section 3F of the man pages).

The system errors generated by the operating system are described, on C Series machines, in the intro(2) man page. On SPP Series machines, they are explained in the errno(2) man page

Runtime I/O errors (C Series)

The following runtime error messages are generated by the C Series Fortran I/O runtime library:

```
100  error in format
```

See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

```
101  illegal unit number
```

You cannot close logical unit 0. Valid unit numbers are in the range 0 to 255.

`102  formatted I/O not allowed`

The logical unit was opened for unformatted I/O.

`103  unformatted I/O not allowed`

The logical unit was opened for formatted I/O.

`104  direct I/O not allowed`

The logical unit was opened for sequential access, or the logical record length was specified as 0.

`105  sequential I/O not allowed`

The logical unit was opened for direct-access I/O.

`106  can't backspace file`

The file associated with the logical unit can't seek. May be a device or a pipe.

`107  off beginning of record`

The format specified a left tab off the beginning of the record.

`108  can't stat file`

The system cannot return status information about the file. Perhaps the directory is unreadable.

`109  no * after repeat count`

Repeat counts in list-directed I/O must be followed by an asterisk (*) with no blank spaces.

`110  off end of record`

A formatted write tried to go beyond the logical end-of-record. An unformatted read or write also causes this.

`112  incomprehensible list input`

Bad input data for list-directed read.

`113  out of free space`

The library dynamically creates buffers for internal use. Not enough memory was available at the time of the request.

`114   unit not connected`

The unit was not open.

`115   read unexpected character`

Certain format conversions cannot tolerate nonnumeric data. Logical data must be `T` or `F`.

`116   blank logical input field`

`117   'new' file exists`

You tried to open an existing file with `STATUS='NEW'`.

`118   can't find 'old' file`

You tried to open a nonexistent file with `STATUS='OLD'`.

`119   unknown system error`

Contact the CONVEX Technical Assistance Center (TAC). See the "How to use this guide" preface for information on doing this.

`120   requires seek ability`

Direct-access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

`121   illegal argument`

Certain arguments to `OPEN`, and other I/O statements, are checked for legitimacy.

`122   negative repeat count`

The repeat count for list-directed input must be a positive integer.

`123   illegal operation for channel or device`

`124   new record not allowed`

Encode and decode can only read and write single records.

`125 numeric keyword variable overflowed`

A keyword variable such as ASSOCIATEVARIABLE overflowed.

`126 record number greater than max records`

A direct-access was attempted to a record number less than one or greater than MAXREC specified in the OPEN statement.

`127 file is read-only`

Writing is not permitted to file opened with the READONLY keyword.

`128 variable record type not allowed`

Direct-access files may not have variable-length records.

`129 exceeded record length`

A read was attempted past the end of a record in a sequentially accessed file with RECL set on OPEN.

`130 exceeds maximum number of open files`

A maximum of 255 files may be open at one time. This is a system-dependent limit.

`131 data type size too small for REAL format code`

Format code of variables less than 4 bytes cannot be read or written with the E, O, F, or G format code.

`132 infinite loop in format`

`133 fixed record type not allowed for print files`

`134 attempt to read nonexistent record in a direct access file`

Returned for direct-access reads when an attempt is made to read a record that does not exist.

135 reopening file with different unit number not allowed

136 I/O list item format code mismatch

137 unknown record length

A record length must be specified for the file.

138 async I/O not allowed

139 sync I/O not allowed

140 incompatible format structure

The internal representation of parsed format strings has changed. The routine must be recompiled.

141 namelist error

An error has been detected in the use of namelist-directed I/O.

142 suspect recursive logical name definition

143 recursive file I/O.

144 out of free, could be UFIO on a formatted file

145 String to numb convert error

146 data format conversion routine returned an err

An error has been detected in the use of binary I/O.

147 illegal request for partial record I/O (bufferin/out)

148 invalid write after ENDFILE statement

149 invalid internal I/O record size

Runtime I/O errors (SPP Series)

The following runtime error messages are generated by the SPP Series Fortran I/O runtime library.

```
900   general error in format
```

See error message output for the location of the error in the forrmat. Can be caused by more than 10 levels of nested parentheses, or an extremely long format statement.

```
901   illegal file unit number less than zero
```

Valid unit numbers are in the range 0 to 255.

```
902   formatted I/O attempted on unformatted
      file
```

The logical unit was opened for unformatted I/O.

```
903   unformatted I/O attempted on formatted
      file
```

The logical unit was opened for formatted I/O.

```
904   direct I/O attempted on sequential-only
      file/device
```

The logical unit was opened for sequential access, or the logical record length was specified as 0.

```
905   list-directed input incompatible with
      log variable
```

The logical unit was opened for direct-access I/O.

```
906   BACKSPACE, ENDFILE, or REWIND attempted
      on a tty
```

The file associated with the logical unit can't seek. May be a device or a pipe.

```
907   illegal character starting list-directed
      read item
```

The format specified a left tab off the beginning of the record.

908 file system wouldn't open file requested

909 BACKSPACE, ENDFILE, or REWIND attempted on a device or direct file for which these operations are undefined

910 access past end-of-record attempted

A formatted write tried to go beyond the logical end-of-record. An unformatted read or write also causes this.

912 invalid complex number on input

913 out of free space in the heap

The library dynamicallly creates buffers for internal use. Not enough memory was available at the time of the request.

914 I/O attempted on unopened unit number

The unit was not open.

915 illegal character read for this type of input

Certain format conversions cannot tolerate nonnumeric data. Logical data must be T or F.

916 type of input incompatible with BOOLEAN variable in I/O list

917 OPEN: name provided for scratch file

918 status specified as new but file exists already

919 ACCESS=KEYED but KEY=() is missing on OPEN

920 OPEN attempted of file already open with another unit

921 OPEN: BLANK specification not consistent with FORM specification

922 directly accessed missing record -- past EOF

923 FORM specifier in OPEN conflicts with previous

924 CLOSE of scratch file with KEEP specified

925 OPEN: invalid STATUS specification

926 CLOSE STATUS specifier passed was undefined

927 access method in OPEN conflicts with previous

929 OPEN: open of direct file with no RECL specifier

930 OPEN: invalid record length specification

931 OPEN: invalid BLANK specification

933 end of file found when END= parameter not used

934 OPEN: READONLY not consistent with APPEND access

936 append I/O attempted on sequential-only file or device

937 direct-access of record numbered <= 0 requested

942 string assigned to a non-character on input

944 error in direct unformatted I/O

945 error in formatted I/O -- usually means more characters requested than exist in record

950 array or substring index out of range

951 assigned GOTO label did not correspond to list

952 DO-loop increment was zero

953 no repeatable edit descriptor in format

954 FORMAT () used with I/O list items present

955 OPEN: file specifier not consistent with STATUS specifier

956 general file system error -- file system message also

957 FORMAT descriptor incompatible with numeric variable in I/O list

958 FORMAT descriptor incompatible with CHARACTER variable in I/O list

959 FORMAT descriptor incompatible with LOGICAL variable in I/O list

960 missing starting left parenthesis

961 invalid FORMAT descriptor

962 I,F,E,D,G,L,A,P,H,X or (expected.

963 trying to scale unscalable specifier

964 nesting of parentheses too deep

965 invalid TAB specifier

966 invalid BLANK specifier

967 specifier expected but end of format found (or: unbalanced left parenthesis in format)

968 invalid FORMAT descriptor separator missing -- expected ",", ":", "/", or ")"

969 digit expected but not found

970 period expected in floating point FORMAT descriptor

971 unbalanced parenthesis

972 string too long or extends past end of format

973 record length specified in OPEN conflicts

974 EOF on internal "file"

975 illegal value requested for new O/S file number

976 unexpected character in NAMELIST read

977 illegal subscript or substring in NAMELIST read

978 too many values in NAMELIST read

979 variable not in NAMELIST group in NAMELIST read

980 NAMELIST I/O on unformatted file

981 value read for numeric item is too big or small

988 program has ISAM file usage, should be linked with HP-ISAM library using -lisam option

994 RECORDTYPE= variable not supported

995 program has ISAM file usage, not supported on this architecture

996 error when scanning exponent

999 error in FOR_nnnOPEN environment variable

Runtime libraries

E

This appendix describes the Fortran runtime libraries. Both the CONVEX C Series and CONVEX SPP Series runtime libraries are described here. The names and contents of the standard Fortran runtime libraries are listed in Table 39 and Table 40.

Any libraries specified on the compiler command (`fc`) line with the `-l` loader option (specified as `-Wl,-l`) are searched before the standard libraries. The `libI66.a` library is required only for FORTRAN 66 compatibility. This library is included automatically if the `-F66` compiler option is specified.

A library linked with the `fc` driver cannot contain a main program written in C.

The `libU77.a` library functions are also available. See the intro(3f) man page for a description of these functions.

Table 39 lists the runtime libraries available for use with CONVEX Fortran on C Series machines. Table 40 lists CONVEX Fortran's SPP Series libraries.

Table 39 Fortran runtime libraries (C Series)

Library name	Contents
/usr/lib/libF77.a	Intrinsic function library
/usr/lib/libF77_p.a	Profiled intrinsic function library
/usr/lib/palib/libF77.a	CXpa-profiled libF77.a*
/usr/lib/libF90.a	Fortran 90 intrinsic function library
/usr/lib/libF90_p.a	Profiled Fortran 90 intrinsic function library
/usr/lib/palib/libF90.a	CXpa-profiled libF90.a*
/usr/lib/libI77.a	Fortran I/O library
/usr/lib/libI77_p.a	Profiled Fortran I/O library
/usr/lib/palib/libI77.a	CXpa-profiled libI77.a*
/usr/lib/libU77.a	ConvexOS interface library
/usr/lib/libU77_p.a	Profiled ConvexOS interface library
/usr/lib/palib/libU77.a	CXpa-profiled ConvexOS interface library*
/usr/lib/libU77p8.a	Equivalent to libU77.a except uses -p8 default variable lengths. See Chapter 1 of this guide for more information on the -p8 compiler option.
/usr/lib/libU77p8_p.a	Equivalent to libU77p8.a but for use with -p option.
/usr/lib/palib/libU77p8.a	CXpa-profiled libU77p8.a*
/usr/lib/libU77pd8.a	Equivalent to libU77.a except uses -pd8 default variable lengths. See Chapter 1 of this guide for more information on the -pd8 compiler option.
/usr/lib/libU77pd8_p.a	Equivalent to libU77pd8.a but for use with -p option.
/usr/lib/palib/libU77pd8.a	CXpa profiled libU77pd8.a*
/usr/lib/liblfs.a	ConvexOS Version 10.0-specific library
/usr/lib/libI66.a	Fortran 66 I/O initialization
/usr/lib/libV77.a	Constants for VAX Fortran compatibility
/usr/lib/libvfn.a	VMS-to-ConvexOS file name translation routines
/usr/lib/libD77.a	Dummy VMS-to-ConvexOS file name translation routines
/usr/lib/libc.a	Extended ANSI C library (system utilities)
/usr/lib/libc_old.a	Backward compatible libc.a

Table 39 **(continued)** Fortran runtime libraries (C Series)

Library name	Contents
/usr/lib/libc_p.a	Profiled C library
/usr/lib/libc_old_p.a	Backward compatible `libc_p.a`
/usr/lib/libm.a	Math library
/usr/lib/libmathC1.a	Math library optimized for C1 architecture
/usr/lib/libmathC1_p.a	Profiled math library optimized for C1 architecture
/usr/lib/libmathC2.a	Math library optimized for C2 architecture
/usr/lib/libmathC2_p.a	Profiled math library optimized for C2 architecture
/usr/lib/libcfc.a	Cray compatibility library
/usr/lib/libcfc_p.a	Profiled Cray compatibility library
/usr/lib/palib/libcfc.a	CXpa profiled Cray compatibility library

*CXpa is an optional product. CXpa-profiled libraries may not be installed on your system or `-cxpa` libraries may be symbolically linked to the unprofiled versions. `-cxpa` libraries are present in the `palib` subdirectory of the directory containing the other libraries; this is normally `/usr/lib`.

Table 40 lists SPP Series Fortran runtime libraries. These libraries reside in the `/usr/convex/fc`*Ver*`/lib` directory, where *Ver* is the version number of the Fortran compiler. To obtain the version number of the CONVEX Fortran compiler, use `-vn` compiler option.

Table 40 Fortran runtime libraries (SPP Series)

Library name	Contents
/usr/convex/fc*Ver*/lib/libF90.a	Fortran 90 intrinsic function library
/usr/convex/fc*Ver*/lib/libU77.a	SPP-UX interface library
/usr/convex/fc*Ver*/lib/spp1/libcl.a	Intrinsics and other supported routines
/usr/convex/fc*Ver*/lib/crt0.o	Startup routines
/usr/convex/fc*Ver*/lib/libc.a	Extended ANSI C library

Intrinsic library and math library

This section summarizes the runtime routine entry points in the Fortran Intrinsic Library and the CONVEX Math Library. These libraries include runtime routines for Fortran intrinsics, mathematical programmed operators, and character-string programmed operators. They are loaded automatically by `fc`.

Calling conventions

The CONVEX Math Library runtime routines are accessible to all language processors. The functions must be called using the `callq/rtnq` mechanism (in assembly language) with arguments passed by value in the scalar or vector registers and function results returned in the appropriate type register(s). Runtime routines that accept multiple vector arguments, such as `COMPLEX` division routines, restrict all arguments to the same length, and the length of the resultant vector is the same as the argument vector length.

`COMPLEX` and `REAL*16` values are represented as pairs of registers. In the case of `COMPLEX` values, the real part resides in the low-order register and the imaginary part in the high-order register. `REAL*16` values are available on SPP Series machines and in native mode on C Series machines.

C Series only

On C Series machines, math intrinsics that take vector arguments or return vector results are passed in vector registers.

The functions in `libF77` (on C Series) or `libcl.a` (on SPP Series) use the standard Fortran calling conventions. For many intrinsics, there are entry points in the intrinsic library and the math library. The `libF77` entry points are provided for compatibility with the standard Fortran calling convention and must be used when intrinsics are referenced as dummy arguments. In most cases, `libF77` routines do not perform the intrinsic operation but call the corresponding CONVEX math runtime routine. Not all intrinsics require runtime routines. For example, scalar truncation can be accomplished with a single convert instruction, and, as such, is implemented as inline code.

Function-naming convention (C Series only)

The runtime routine names are constructed as follows:

`{mth$ | for$}`*<argument-type(s)>_<function-type>_<result-type>*

Scalar intrinsics have entry points in the Fortran Intrinsic Library (`for$` prefix) and the CONVEX Math Library (`mth$`

prefix); vector intrinsics have entry points only in the math library.

The *function-type* is typically the generic intrinsic name. For example, mth$d_sqrt is the REAL*8 scalar square root entry point in the CONVEX Math Library.

If the *argument-type* and *result-type* are the same, the *result-type* is omitted from the function name. If the function has multiple arguments all of the same type, then a single argument code is used rather than a sequence of codes. The codes are given in Table 41.

Table 41 Argument/result codes

Code	Type of argument/result
h	INTEGER*1
i	INTEGER*2
j	INTEGER*4
k	INTEGER*8
l	LOGICAL*4
r	REAL*4
c	COMPLEX*8
d	REAL*8
q	REAL*16
z	COMPLEX*16
s	CHARACTER*N
v	Vector (C Series only)
u	Unsigned
s	String
vm	Vector mask register

When multiple arguments of different types are used, the order of the arguments conforms to the intrinsic definition. For example, the scalar/vector KISHIFT intrinsic is implemented with the runtime routine mth$jvj_shft. This runtime routine performs a logical shift of an INTEGER*4 scalar by an INTEGER*4 vector and returns the result as an INTEGER*4 vector.

Example:

```
DO I=1,N
   K(I) = KISHFT(L,M(I))
ENDDO
```

Intrinsic runtime routines

Table 42 summarizes calling sequences for the intrinsic runtime routines. Braces { } indicate prefixes that are available for each runtime routine. Specific intrinsic names are provided for cross-reference. There is not always a one-to-one correspondence between intrinsic references and runtime routines. In some cases, two different intrinsics (separated by commas) generate a call to the same runtime routine, or the application of multiple intrinsics (denoted by the use of parentheses) generates a call to a single runtime routine. Some runtime routines listed in this table are used as programmed operators. For example, the assignment of an INTEGER*4 vector to a REAL*4 vector generates a call to mth$vj_cvt_vr

Table 42 Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Square root				
SQRT	{for,mth}$r_sqrt	FTN_SQRT	r	r
	mth$vr_sqrt		vr	vr
DSQRT	{for,mth}$d_sqrt	FTN_DSQRT	d	d
	mth$vd_sqrt		vd	vd
CSQRT	{for,mth}$c_sqrt	FTN_CSQRT	c	c
	mth$vc_sqrt		vc	vc
CDSQRT	{for,mth}$z_sqrt	ZSQRT	z	z
	mth$vz_sqrt		vz	vz
QSQRT	{for,mth}$q_sqrt	FTN_QSQRT	q	q
	mth$vq_sqrt		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Natural logarithm				
LOG	{for,mth}$r_log	FTN_ALOG	r	r
	mth$vr_log		vr	vr
DLOG	{for,mth}$d_log	FTN_DLOG	d	d
	mth$vd_log		vd	vd
CLOG	{for,mth}$c_log	FTN_CLOG	c	c
	mth$vc_log		vc	vc
CDLOG	{for,mth}$z_log	ZLOG	z	z
	mth$vz_log		vz	vz
QLOG	{for,mth}$q_log	FTN_QLOG	q	q
	mth$vq_log		vq	vq
Common logarithm				
LOG10	{for,mth}$r_log10	FTN_ALOG10	r	r
	mth$vr_log10		vr	vr
DLOG10	{for,mth}$d_log10	FTN_DLOG10	d	d
	mth$vd_log10		vd	vd
QLOG10	{for,mth}$q_log10	FTN_QLOG10	q	q
	mth$vq_log10		vq	vq
Exponential				
EXP	{for,mth}$r_exp	FTN_EXP	r	r
	mth$vr_exp		vr	vr
DEXP	{for,mth}$d_exp	FTN_DEXP	d	d
	mth$vd_exp		vd	vd
CEXP	{for,mth}$c_exp	FTN_CEXP	c	c
	mth$vc_exp		vc	vc
CDEXP	{for,mth}$z_exp	ZEXP	z	z
	mth$vz_exp		vz	vz
QEXP	{for,mth}$q_exp	FTN_QEXP	q	q
	mth$vq_exp		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Sine				
SIN	{for,mth}$r_sin	FTN_SIN	r	r
	mth$vr_sin		vr	vr
DSIN	{for,mth}$d_sin	FTN_DSIN	d	d
	mth$vd_sin		vd	vd
CSIN	{for,mth}$c_sin	FTN_CSIN	c	c
	mth$vc_sin		vc	vc
CDSIN	{for,mth}$z_sin	ZSIN	z	z
	mth$vz_sin		vz	vz
QSIN	{for,mth}$q_sin	FTN_QSIN	q	q
	mth$vq_sin		vq	vq
Sine (degree)				
SIND	{for,mth}$r_sind	FTN_SIND	r	r
	mth$vr_sind		vr	vr
DSIND	{for,mth}$d_sind	FTN_DSIND	d	d
	mth$vd_sind		vd	vd
QSIND	{for,mth}$q_sind	FTN_QSIND	q	q
	mth$vq_sind		vq	vq
Cosine				
COS	{for,mth}$r_cos	FTN_COS	r	r
	mth$vr_cos		vr	vr
DCOS	{for,mth}$d_cos	FTN_DCOS	d	d
	mth$vd_cos		vd	vd
CCOS	{for,mth}$c_cos	FTN_CCOS	c	c
	mth$vc_cos		vc	vc
CDCOS	{for,mth}$z_cos	ZCOS	z	z
	mth$vz_cos		vz	vz
QCOS	{for,mth}$q_cos	FTN_QCOS	q	q
	mth$vq_cos		vq	vq
Cosine (degree)				
COSD	{for,mth}$r_cosd	FTN_ACOSD	r	r
	mth$vr_cosd		vr	vr
DCOSD	{for,mth}$d_cosd	FTN_DCOSD	d	d
	mth$vd_cosd		vd	vd
QCOSD	{for,mth}$q_cosd	FTN_QCOSD	q	q
	mth$vq_cosd		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Tangent				
TAN	{for,mth}$r_tan	FTN_TAN	r	r
	mth$vr_tan		vr	vr
DTAN	{for,mth}$d_tan	FTN_DTAN	d	d
	mth$vd_tan		vd	vd
QTAN	{for,mth}$q_tan	FTN_QTAN	q	q
	mth$vq_tan		vq	vq
Tangent (degree)				
TAND	{for,mth}$r_tand	FTN_TAND	r	r
	mth$vr_tand		vr	vr
DTAND	{for,mth}$d_tand	FTN_DTAND	d	d
	mth$vd_tand		vd	vd
QTAND	{for,mth}$q_tand	FTN_QTAND	q	q
	mth$vq_tand		vq	vq
Arc sine				
ASIN	{for,mth}$r_asin	FTN_ASIN	r	r
	mth$vr_asin		vr	vr
DASIN	{for,mth}$d_asin	FTN_DASIN	d	d
	mth$vd_asin		vd	vd
QASIN	{for,mth}$q_asin	FTN_QASIN	q	q
	mth$vq_asin		vq	vq
Arc sine (degree)				
ASIND	{for,mth}$r_asind	FTN_ASIND	r	r
	mth$vr_asind		vr	vr
DASIND	{for,mth}$d_asind	FTN_DASIND	d	d
	mth$vd_asind		vd	vd
QASIND	{for,mth}$q_asind	FTN_QASIND	q	q
	mth$vq_asind		vq	vq
Arc cosine				
ACOS	{for,mth}$r_acos	FTN_ACOS	r	r
	mth$vr_acos		vr	vr
DACOS	{for,mth}$d_acos	FTN_DACOS	d	d
	mth$vd_acos		vd	vd
QACOS	{for,mth}$q_acos	FTN_QACOS	q	q
	mth$vq_acos		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Arc cosine (degree)				
ACOSD	{for,mth}$r_acosd	FTN_ACOSD	r	r
	mth$vr_acosd		vr	vr
DACOSD	{for,mth}$d_acosd	FTN_DACOSD	d	d
	mth$vd_acosd		vd	vd
QACOSD	{for,mth}$q_acosd	FTN_QACOSD	q	q
	mth$vq_acosd		vq	vq
Arc tangent				
ATAN	{for,mth}$r_atan	FTN_ATAN	r	r
	mth$vr_atan		vr	vr
DATAN	{for,mth}$d_atan	FTN_DATAN	d	d
	mth$vd_atan		vd	vd
QATAN	{for,mth}$q_atan	FTN_QATAN	q	q
	mth$vq_atan		vq	vq
Arc tangent with two arguments				
ATAN2	{for,mth}$r_atan2	FTN_ATAN2	r,r	r
	mth$vr_atan2		vr,vr	vr
	mth$vrr_atan2		vr,r	vr
	mth$rvr_atan2		r,vr	vr
DATAN2	{for,mth}$d_atan2	FTN_DATAN2	d,d	d
	mth$vd_atan2		vd,vd	vd
	mth$vdd_atan2		vd,d	vd
	mth$dvd_atan2		d,vd	vd
QATAN2	{for,mth}$q_atan2	FTN_QATAN2	q,q	q
	mth$vq_atan2		vq,vq	vq
Arc tangent (degree)				
ATAND	{for,mth}$r_atand	FTN_ATAND	r	r
	mth$vr_atand		vr	vr
DATAND	{for,mth}$d_atand	FTN_DATAND	d	d
	mth$vd_atand		vd	vd
QATAND	{for,mth}$q_atand	FTN_QATAND	q	q
	mth$vq_atand		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Arc tangent with two arguments (degree)				
ATAN2D	{for,mth}$r_atan2d	FTN_ATAN2D	r,r	r
	mth$vr_atan2d		vr,vr	vr
	mth$vrr_atan2d		vr,r	vr
	mth$rvr_atan2d		r,vr	vr
DATAN2D	{for,mth}$d_atan2d	FTN_DATAN2D	d,d	d
	mth$vd_atan2d		vd,vd	vd
	mth$vdd_atand2		vd,d	vd
	mth$dvd_atand2		d,vd	vd
QATAN2D	{for,mth}$q_atan2d	FTN_QATAN2D	q,q	q
	mth$vq_atan2d		vq,vq	vq
Hyperbolic sine				
SINH	{for,mth}$r_sinh	FTN_SINH	r	r
	mth$vr_sinh		vr	vr
DSINH	{for,mth}$d_sinh	FTN_DSINH	d	d
	mth$vd_sinh		vd	vd
QSINH	{for,mth}$q_sinh	FTN_QSINH	q	q
	mth$vq_sinh		vq	vq
Hyperbolic cosine				
COSH	{for,mth}$r_cosh	FTN_COSH	r	r
	mth$vr_cosh		vr	vr
DCOSH	{for,mth}$d_cosh	FTN_DCOSH	d	d
	mth$vd_cosh		vd	vd
QCOSH	{for,mth}$q_cosh	FTN_QCOSH	q	q
	mth$vq_cosh		vq	vq
Hyperbolic tangent				
TANH	{for,mth}$r_tanh	FTN_TANH	r	r
	mth$vr_tanh		vr	vr
DTANH	{for,mth}$d_tanh	FTN_DTANH	d	d
	mth$vd_tanh		vd	vd
QTANH	{for,mth}$q_tanh	FTN_QTANH	q	q
	mth$vq_tanh		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Float-to-fix conversion				
INT1(IINT)	mth$vr_cvt_vh		vr	vh
IINT,IIFIX	{for,mth}$r_cvt_i		r	i
	mth$vr_cvt_vi		vr	vi
JINT,JIFIX	{for,mth}$r_cvt_j		r	j
	mth$vr_cvt_vj		vr	vj
KINT,KIFIX	{form,mth}$r_cvt_k		r	k
	mth$vr_cvt_vk		vr	vk
INT1(IIDINT)	mth$vd_cvt_vh		vd	vh
IIDINT,IIDINT	{for,mth}$d_cvt_i		d	i
	mth$vd_cvt_vi		vd	vi
JIDINT,JIDINT	{for,mth}$d_cvt_j		d	j
	mth$vd_cvt_vj		vd	vj
KIDINT,KIDINT	{for,mth}$d_cvt_k		d	k
	mth$vd_cvt_vk		vd	vk
	{for,mth}$q_cvt_h		q	h
	mth$vq_cvt_vh		vq	vh
IIQUINT	{for,mth}$q_cvt_i		q	i
	mth$vq_cvt_vi		vq	vi
JIQINT	{for,mth}$q_cvt_j		q	j
	mth$vq_cvt_vj		vq	vj
KIQINT	{for,mth}$q_cvt_k		q	k
	mth$vq_cvt_vk		vq	vk

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Fix-to-float conversion				
FLOATI(INT2)	mth$vh_cvt_vr		vh	vr
DFLOTI(INT2)	mth$vh_cvt_vd		vh	vd
QFLOATI(INT2)	mth$h_cvt_q		h	q
	mth$vh_cvt_vq		vh	vq
FLOATI	{for,mth}$i_cvt_r		i	r
	mth$vi_cvt_vr		vi	vr
DFLOTI	{for,mth}$i_cvt_d		i	d
	mth$vi_cvt_vd		vi	vd
QFLOTI	{for,mth}$i_cvt_q		i	q
	mth$vi_cvt_vq		vi	vq
FLOATJ	{for,mth}$j_cvt_r		j	r
	mth$vj_cvt_vr		vj	vr
DFLOTJ	{for,mth}$j_cvt_d		j	d
	mth$vj_cvt_vd		vj	vd
QFLOTJ	{for,mth}$j_cvt_q		j	q
	mth$vj_cvt_vq		vj	vq
FLOATK	{for,mth}$k_cvt_r		k	r
	mth$vk_cvt_vr		vk	vr
DFLOTK	{for,mth}$k_cvt_d		k	d
	mth$vk_cvt_vd		vk	vd
QFLOTK	{for,mth}$k_cvt_q		k	q
	mth$vk_cvt_vq		vk	vq
Integer part of real				
AINT	{for,mth}$r_int	FTN_AINT	r	r
	mth$vr_int		vr	vr
DINT	{for,mth}$d_int	FTN_DINT	d	d
	mth$vd_int		vd	vd
QINT	{for,mth}$q_int	FTN_QINT	q	q
	mth$vq_int		vq	vq
Real part of complex				
REAL,SNGL	for$c_real		c	c
DREAL,DBLE	for$z_real		z	z
Imaginary part of complex				
AIMAG	for$c_imag		c	c
DIMAG	for$z_imag		z	z

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Complex conjugate				
CONJG	for$c_conj		c	c
DCONJG	for$z_conj		z	z
Maximum (pairwise operation, not a reduction)				
IMAX0	mth$vi_max		vi,vi	vi
	mth$vii_max		vi,i	vi
JMAX0	mth$vj_max		vj,vj	vj
	mth$vjj_max		vj,j	vj
	for$j_imax		j,j	j
KMAX0	mth$vk_max		vk,vk	vk
	mth$vkk_max		vk,k	vk
AMAX1	mth$vr_max		vr,vr	vr
	mth$vrr_max		vr,r	vr
	for$r_imax		r,r	r
DMAX1	mth$vd_max		vd,vd	vd
	mth$vdd_max		vd,d	vd
	for$d_imax		d,d	d
QMAX1	mth$q_max		q,q	q
	mth$vq_max		vq,vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Minimum (pairwise operation, not a reduction)				
IMINO	mth$vi_min		vi,vi	vi
	mth$vii_min		vi,i	vi
JMINO	mth$vj_min		vj,vj	vj
	mth$vjj_min		vj,j	vj
	for$j_imin		j,j	j
KMINO	mth$vk_min		vk,vk	vk
	mth$vkk_min		vk,k	vk
AMIN1	mth$vr_min		vr,vr	vr
	mth$vrr_min		vr,r	vr
	for$r_imin		r,r	r
DMIN1	mth$vd_min		vd,vd	vd
	mth$vdd_min		vd,d	vd
	for$d_imin		d,d	d
QMIN1	mth$q_min		q,q	q
	mth$vq_min		vq,vq	vq
REAL*8 product of REAL*4				
DPROD	{for,mth}$r_prod_d		r	d
	mth$vr_prod_vd		vr	vd
	mth$vrr_prod		vr,r	vd

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Positive difference				
IIDIM	{for,mth}$i_dim		i,i	i
	mth$vi_dim		vi,vi	vi
	mth$vii_dim		vi,i	vi
	mth$ivi_dim		i,vi	vi
JIDIM	{for,mth}$j_dim		j,j	j
	mth$vj_dim		vj,vj	vj
	mth$vjj_dim		vj,j	vj
	mth$jvj_dim		j,vj	vj
KIDIM	{for,mth}$k_dim		k,k	k
	mth$vk_dim		vk,vk	vk
	mth$vkk_dim		vk,k	vk
	mth$kvk_dim		k,vk	vk
DIM	{for,mth}$r_dim		r,r	r
	mth$vr_dim		vr,vr	vr
	mth$vrr_dim		vr,r	vr
	mth$rvr_dim		r,vr	vr
DDIM	{for,mth}$d_dim		d,d	d
	mth$vd_dim		vd,vd	vd
	mth$vdd_dim		vd,d	vd
	mth$dvd_dim		d,vd	vd
QDIM	{for,mth}$q_dim		q,q	q
	mth$vq_dim		vq,vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Remainder				
IMOD	{for,mth}$i_mod		i,i	i
	mth$vi_mod		vi,vi	vi
	mth$vii_mod		vi,i	vi
	mth$ivi_mod		i,vi	vi
JMOD	{for,mth}$j_mod		j,j	j
	mth$vj_mod		vj,vj	vj
	mth$vjj_mod		vj,j	vj
	mth$jvj_mod		j,vj	vj
KMOD	{for,mth}$k_mod	FTN_KMOD	k,k	k
	mth$vk_mod		vk,vk	vk
	mth$vkk_mod		vk,k	vk
	mth$kvk_mod		k,vk	vk
AMOD	{for,mth}$r_mod	FTN_AMOD	r,r	r
	mth$vr_mod		vr,vr	vr
	mth$vrr_mod		vr,r	vr
	mth$rvr_mod		r,vr	vr
DMOD	{for,mth}$d_mod	FTN_DMOD	d,d	d
	mth$vd_mod		vd,vd	vd
	mth$vdd_mod		vd,d	vd
	mth$dvd_mod		d,vd	vd
QMOD	{for,mth}$q_mod		q,q	q
	mth$vq_mod		vq,vq	vq

Table 42 **(continued)** Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Transfer of sign				
IISIGN	{for,mth}$i_sign	FTN_HSIGN	i,i	i
	mth$vi_sign		vi,vi	vi
	mth$vii_sign		vi,i	vi
	mth$ivi_sign		i,vi	vi
JISIGN	{for,mth}$j_sign	FTN_ISIGN	j,j	j
	mth$vj_sign		vj,vj	vj
	mth$vjj_sign		vj,j	vj
	mth$jvj_sign		j,vj	vj
KISIGN	{for,mth}$k_sign	FTN_KSIGN	k,k	k
	mth$vk_sign		vk,vk	vk
	mth$vkk_sign		vk,k	vk
	mth$kvk_sign		k,vk	vk
SIGN	{for,mth}$r_sign	FTN_SIGN	r,r	r
	mth$vr_sign		vr,vr	vr
	mth$vrr_sign		vr,r	vr
	mth$rvr_sign		r,vr	vr
DSIGN	{for,mth}$d_sign	FTN_DSIGN	d,d	d
	mth$vd_sign		vd,vd	vd
	mth$vdd_sign		vd,d	vd
	mth$dvd_sign		d,vd	vd
QSIGN	{for,mth}$q_sign		q,q	q
	mth$vq_sign		vq,vq	vq
Bitwise AND				
IIAND	{for,mth}$i_and	IAND_2	i,i	i
JIAND	{for,mth}$j_and	IAND_4	j,j	j
KIAND	{for,mth}$k_and	IAND_8	k,k	k
Bitwise OR				
IIOR	{for,mth}$i_or	IOR_2	i,i	i
JIOR	{for,mth}$j_or	IOR_4	j,j	j
KIOR	{for,mth}$k_or	IOR_8	k,k	k
Bitwise XOR				
IIEOR	{for,mth}$i_xor	IEOR_2	i,i	i
JIEOR	{for,mth}$j_xor	IEOR_4	j,j	j
KIEOR	{for,mth}$k_xor	IEOR_8	k,k	k

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Bitwise complement				
INOT	{for,mth}$i_not	NOT_2	i,i	i
JNOT	{for,mth}$j_not	NOT_4	j,j	j
KNOT	{for,mth}$k_not	NOT_8	k,k	k
Bitwise shift				
IISHFT	{for,mth}$i_shft	I_SHFT	i,i	i
	mth$vi_shft		vi,vi	vi
	mth$ivi_shft		i,vi	vi
JISHFT	{for,mth}$j_shft	J_SHFT	j,j	j
	mth$vj_shft		vj,vj	vj
	mth$jvj_shft		j,vj	vj
KISHFT	{for,mth}$k_shft	K_SHFT	k,k	k
	mth$vk_shft		vk,vk	vk
	mth$kvk_shft		k,vk	vk
Bitwise extract				
IIBITS	{for,mth}$i_bits	I_BITS	i,i,i	i
	mth$vi_bits		vi,vi,vi	vi
	mth$vivii_bits		vi,vi,i	vi
	mth$viivi_bits		vi,i,vi	vi
	mth$viii_bits		vi,i,i	vi
JIBITS	{for,mth}$j_bits	J_BITS	j,j,j	j
	mth$vj_bits		vj,vj,vj	vj
	mth$vjvjj_bits		vj,vj,j	vj
	mth$vjjvj_bits		vj,j,vj	vj
	mth$vjjj_bits		vj,j,j	vj
KIBITS	{for,mth}$k_bits	K_BITS	k,k,k	k
	mth$vk_bits		vk,vk,vk	vk
	mth$vkvkk_bits		vk,vk,k	vk
	mth$vkkvk_bits		vk,k,vk	vk
	mth$vkkk_bits		vk,k,k	vk

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Bitwise set				
IIBSET	{for,mth}$i_set	I_BSET	i,i	i
	mth$vi_set		vi,vi	vi
	mth$vii_set		vi,i	vi
	mth$ivi_set		i,vi	vi
JIBSET	{for,mth}$j_set	J_BSET	j,j	j
	mth$vj_set		vj,vj	vj
	mth$vjj_set		vj,j	vj
	mth$jvj_set		j,vj	vj
KIBSET	{for,mth}$k_set	K_BSET	k,k	k
	mth$vk_set		vk,vk	vk
	mth$vkk_set		vk,k	vk
	mth$kvk_set		k,vk	vk
Bitwise test				
BITEST	{for,mth}$i_test	I_BTEST	i,i	i
	mth$vi_test		vi,vi	vi
	mth$vii_test		vi,i	vi
	mth$ivi_test		i,vi	vi
BJTEST	{for,mth}$j_test	J_BTEST	j,j	j
	mth$vj_test		vj,vj	vj
	mth$vjj_test		vj,j	vj
	mth$jvj_test		j,vj	vj
BKTEST	{for,mth}$k_test	K_BTEST	k,k	k
	mth$vk_test		vk,vk	vk
	mth$vkk_test		vk,k	vk
	mth$kvk_test		k,vk	vk
Bitwise clear				
IIBCLR	{for,mth}$i_clr	I_BCLR	i,i	i
	mth$vi_clr		vi,vi	vi
	mth$vii_clr		vi,i	vi
	mth$ivi_clr		i,vi	vi
JIBCLR	{for,mth}$j_clr	J_BCLR	j,j	j
	mth$vj_clr		vj,vj	vj
	mth$vjj_clr		vj,j	vj
	mth$jvj_clr		j,vj	vj
KIBCLR	{for,mth}$k_clr	K_BCLR	k,k	k
	mth$vk_clr		vk,vk	vk
	mth$vkk_clr		vk,k	vk
	mth$kvk_clr		k,vk	vk

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Bitwise circular shift				
IISHFTC	{for,mth}$i_shftc	I_SHFTC	i,i,i	i
	mth$vi_shftc		vi,vi,vi	vi
	mth$vivii_shftc		vi,vi,i	vi
	mth$viivi_shftc		vi,i,vi	vi
	mth$viii_shftc		vi,i,i	vi
JISHFTC	{for,mth}$j_shftc	J_SHFTC	j,j,j	j
	mth$vj_shftc		vj,vj,vj	vj
	mth$vjvjj_shftc		vj,vj,j	vj
	mth$vjjvj_shftc		vj,j,vj	vj
	mth$vjjj_shftc		vj,j,j	vj
KISHFTC	{for,mth}$k_shftc	K_SHFTC	k,k,k	k
	mth$vk_shftc		vk,vk,vk	vk
	mth$vkvkk_shftc		vk,vk,k	vk
	mth$vkkvk_shftc		vk,k,vk	vk
	mth$vkkk_shftc		vk,k,k	vk
String length				
LEN	for$s_len_j		s	j
String index				
INDEX	for$s_index_j		s	j
Character relationals				
LLT	for$s_lt_l	LLT	s	l
LLE	for$s_le_l	LLE	s	l
LGT	for$s_gt_l	LGT	s	l
LGE	for$s_ge_l	LGE	s	l
IEEE/native conversions				
RCVTIR	{for,mth}$r_cvti		r	r
	mth$vr_cvti		vr	vr
DCVTID	{for,mth}$d_cvti		d	d
	mth$vd_cvti		vd	vd
IRCVTR	{for,mth}$r_icvt		r	r
	mth$vr_icvt		vr	vr
IDCVTD	{for,mth}$d_icvt		d	d
	mth$vd_icvt		vd	vd

Table 42 **(continued)** Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Absolute value				
IIABS	{for,mth}$i_abs	HABS	i	i
	mth$vi_abs		vi	vi
JIABS	{for,mth}$j_abs	IABS	j	j
	mth$vj_abs		vj	vj
KIABS	{for,mth}$k_abs	KABS	k	k
	mth$vk_abs		vk	vk
ABS	{for,mth}$r_abs	ABS	r	r
	mth$vr_abs		vr	vr
DABS	{for,mth}$d_abs	DABS	d	d
	mth$vd_abs		vd	vd
CABS	{for,mth}$c_abs_r	FTN_CABS	c	c
	mth$vc_abs_vr		vc	vc
CDABS	{for,mth}$z_abs_d	ZABS	z	z
	mth$vz_abs_vd		vz	vz
QABS	{for,mth}$q_abs	QABS	q	q
	mth$vq_abs		vq	vq

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Nearest integer				
ININT	{for,mth}$r_nint_i		r	i
	mth$vr_nint_vi		vr	vi
JNINT	{for,mth}$r_nint_j	NINT	r	j
	mth$vr_nint_vj		vr	vj
KNINT	{for,mth}$r_nint_k		r	k
	mth$vr_nint_vk		vr	vk
IIDNNT	{for,mth}$d_nint_i		d	i
	mth$vd_nint_vi		vd	vi
JIDNNT	{for,mth}$d_nint_j	IDNINT	d	j
	mth$vd_nint_vj		vd	vj
KIDNNT	{for,mth}$d_nint_k		d	k
	mth$vd_nint_vk		vd	vk
ANINT	{for,mth}$r_nint	ANINT	r	r
	mth$vr_nint		vr	vr
DNINT	{for,mth}$d_nint	DNINT	d	d
	mth$vd_nint		vd	vd
IIQNNT	{for,mth}$q_nint_i		q	i
	mth$vq_nint_vi		vq	vi
JIQNNT	{for,mth}$q_nint_j		q	j
	mth$vq_nint_vj		vq	vj
KIQNNT	{for,mth}$q_nint_k		q	k
	mth$vq_nint_vk		vq	vk
QNINT	{for,mth}$q_nint	QNINT	q	q
	mth$vq_nint		vq	vq
Integer conversion				
INT1	mth$vi_cvt_vh		vi	vh
	mth$vj_cvt_vh		vj	vh
	mth$vk_cvt_vh		vk	vh
INT2	mth$vh_cvt_vi		vh	vi
	mth$vj_cvt_vi		vj	vi
	mth$vk_cvt_vi		vk	vi
INT4	mth$vh_cvt_vj		vh	vj
	mth$vi_cvt_vj		vi	vj
	mth$vk_cvt_vj		vk	vj
INT8	mth$vh_cvt_vk		vh	vk
	mth$vi_cvt_vk		vi	vk
	mth$vj_cvt_vk		vj	vk

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
REAL*4 to REAL*8 conversion DBLE	{for,mth}$r_cvt_d mth$vr_cvt_vd		r vr	d vd
REAL*8 to REAL*4 conversion SNGL	{for,mth}$d_cvt_r mth$vd_cvt_vr		d vd	r vr
REAL*4 to *REAL*16* conversion QEXT	mth$r_cvt_q mth$vr_cvt_vq		r vr	q vq
REAL*8 to REAL*16 conversion QEXTD	mth$d_cvt_q mth$vd_cvt_vq		d vd	q vq
REAL*16 to REAL*8 conversion DBLEQ	{for,mth}$q_cvt_d mth$vq_cvt_vd		q vq	d vd
REAL*16 to REAL*4 conversion. SNGLQ	{for,mth}$q_cvt_r mth$vq_cvt_vr		q vq	r vr

Table 42 (continued) Intrinsic functions

Intrinsic	Runtime routine name (C Series)	Runtime routine name (SPP Series)	Arguments	Result
Fortran 90 intrinsics				
DOT_PRODUCT	f90$dot_product	f90$dot_product	Varies†	Varies†
MATMUL	f90$matmul	f90$matmul	Varies†	Varies†
ALL	f90$all	f90$all	Varies†	Varies†
ANY	f90$any	f90$any	Varies†	Varies†
COUNT	f90$count	f90$count	Varies†	Varies†
MAXVAL	f90$maxval	f90$maxval	Varies†	Varies†
MINVAL	f90$minval	f90$minval	Varies†	Varies†
PRODUCT	f90$product	f90$product	Varies†	Varies†
SUM	f90$sum	f90$sum	Varies†	Varies†
MERGE	f90$merge	f90$merge	Varies†	Varies†
PACK	f90$pack	f90$pack	Varies†	Varies†
SPREAD	f90$spread	f90$spread	Varies†	Varies†
UNPACK	f90$unpack	f90$unpack	Varies†	Varies†
CSHIFT	f90$cshift	f90$cshift	Varies†	Varies†
EOSHIFT	f90$eoshift	f90$eoshift	Varies†	Varies†
TRANSPOSE	f90$transpose	f90$transpose	Varies†	Varies†
MAXLOC	f90$maxloc	f90$maxloc	Varies†	Varies†
MINLOC	f90$minloc	f90$minloc	Varies†	Varies†

†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.

Note

Optimizing code containing type conversions (including implicit type conversions) can cause some code to vectorize poorly or not at all.

Exponentiation programmed operators

The runtime routines listed in Table 43 perform exponentiation (**). The Arguments column lists the base type, followed by the exponent type.

Table 43 Exponentiation routines

C Series runtime routine name	SPP Series routine name	Arguments	Result
mth$j_pow	FTN_ITOI	j,j	j
mth$k_pow	FTN_KTOK	k,k	k
mth$r_pow	FTN_RTOR	r,r	r
mth$d_pow	FTN_DTOD	d,d	d
mth$c_pow	FTN_CTOC	c,c	c
mth$z_pow	FTN_ZTOZ	z,z	z
mth$rj_pow_r	FTN_RTOI	r,j	r
mth$dj_pow_d	FTN_DTOI	d,j	d
mth$cj_pow_c	FTN_CTOI	c,j	c
mth$zj_pow_z	FTN_ZTOI	z,j	z
mth$vj_pow		vj,vj	vj
mth$vk_pow		vk,vk	vk
mth$vr_pow		vr,vr	vr
mth$vd_pow		vd,vd	vd
mth$vc_pow		vc,vc	vc
mth$vz_pow		vz,vz	vz
mth$vrvj_pow_vr		vr,vj	vr
mth$vdvj_pow_vd		vd,vj	vd
mth$vcvj_pow_vc		vc,vj	vc
mth$vzvj_pow_vz		vz,vj	vz
mth$vrj_pow_vr		vr,j	vr
mth$vdj_pow_vd		vd,j	vd
mth$vcj_pow_vc		vc,f	vc
mth$vzj_pow_vz		vz,j	vz
mth$jvj_pow		j,vj	vj
mth$kvk_pow		k,vk	vk
mth$rvr_pow		r,vr	vr
mth$dvd_pow		d,vd	vd
mth$cvc_pow		c,vc	vc
mth$zvz_pow		z,vz	vz
mth$rvj_pow_vr		r,vj	vr
mth$dvj_pow_vd		d,vj	vd
mth$cvj_pow_vc		c,vj	vc
mth$zvj_pow_vz		z,vj	vz
mth$q_pow	for$q_pow	q,q	q
mth$vq_pow		vq,vq	vq
mth$qj_pow_q		q,j	q
mth$vqj_pow_vq		vq,vj	vq

Complex programmed operators

The runtime routines listed in Table 44 perform complex multiplication and division. For division, the argument types are listed as "dividend, divisor."

Table 44 Complex programmed operators

Function	C Series runtime routine name	Argument	Result
Complex division	mth$c_div	c,c	c
	mth$z_div	z,z	z
	mth$vc_div	vc,vc	vc
	mth$vz_div	vz,vz	vz
	mth$vcc_div	vc,c	vc
	mth$vzz_div	vz,z	vz
	mth$cvc_div	c,vc	vc
	mth$zvz_div	z,vz	vz
Complex multiplication	mth$c_mul	c,c	c
	mth$z_mul	z,z	z
	mth$vc_mul	vc,vc	vc
	mth$vz_mul	vz,vz	vz
	mth$vcc_mul	vc,c	vc
	mth$vzz_mul	vz,z	vz
	mth$cvc_mul	c,vc	vc
	mth$zvz_mul	z,vz	vz

REAL*16 programmed operators

The runtime routines listed in Table 45 perform comparison and arithmetic functions. The REAL*16 data type is available on SPP Series machines and in native mode on C Series machines.

Table 45 REAL*16 programmed operators

Intrinsic	C Series runtime routine name	SPP Series runtime routine name	Arguments	Result
Comparison operators				
.LE.	{for,mth}$q_le_l mth$vq_le_l	for$q_le_l	q,q vq,vq	l vl
.LT.	{for,mth}$q_lt_l mth$vq_lt_l	for$q_lt_l	q,q vq,vq	l vl
.GE.	{for,mth}$q_ge_l mth$vq_ge_l	for$q_ge_l	q,q vq,vq	l vl
.GT.	{for,mth}$q_gt_l mth$vq_gt_l	for$q_gt_l	q,q vq,vq	l vl
.EQ.	{for,mth}$q_eq_l mth$vq_eq_l	for$q_eq_l	q,q vq,vq	l vl
.NE.	{for,mth}$q_ne_l mth$vq_ne_l	for$q_ne_l	q,q vq,vq	l vl
Arithmetic operators				
*	{for,mth}$q_mul mth$vq_mul	for$q_mul	q vq,vq	q vq
/	{for,mth}$q_div mth$vq_div	for$q_div	q vq,vq	q vq
+	{for,mth}$q_add mth$vq_add	for$q_add	q vq,vq	q vq

Table 45 (continued) REAL*16 programmed operators

Intrinsic	C Series runtime routine name	SPP Series runtime routine name	Arguments	Result
-	{for,mth}$q_sub mth$vq_sub	for$q_sub	q vq,vq	q vq
- (unary)	{for,mth}$q_neg mth$vq_neg	for$q_neg	q vq	q vq

Vector mask programmed operators (C Series)

The C Series runtime routine mth$vm_lastnz finds the position of the highest-order nonzero bit of the vector mask register.

This runtime routine is useful for conditional code, such as:

```
DO I=1,N
  IF(A(I)) B = A(I)
ENDDO
```

This routine is available only on CONVEX C Series machines.

String-manipulation programmed operators

The following are the C Series and SPP Series string-manipulation programmed operators:

C Series operator	SPP Series operator	Description
for$s_cat		String concatenation
for$s_copy		String copy
for$s_stop		STOP runtime routine
for$s_paus		PAUSE runtime routine
for$s_rnge		Subscript out of range report
for$s_eq_l	F$leq_nls	String equal comparison
for$s_ne_l	F$lne_nls	String not equal comparison

Runtime routine data items (C Series)

The runtime routines described in this section are associated with data values used by the runtime routine libraries and the Fortran compiler.

On C Series machines, the label `mth$vmones`, which is two long words of all ones (`1s`), is used to load the vector mask register. No equivalent instruction is available for SPP Series machines, which have no vector registers.

The values -2 through 130 are available in 6 data types:

Data type	C Series runtime library routine
INTEGER*1	mLth$h_indx
INTEGER*2	mth$i_indx
INTEGER*4	mth$j_indx
INTEGER*8	mth$k_indx
REAL*4	mth$r_indx
REAL*4	mth$r_indx_i
REAL*8	mth$d_indx
REAL*8	mth$d_indx_i

The following example shows a typical runtime routine code:

```
DO I = 1, N
   J(I) = I -1
ENDDO
```

Fortran I/O library

This section summarizes the runtime routine entry points in the Fortran I/O library, either `libI77.a` (on C Series) or `libcl.a` (on SPP Series). The Fortran compiler generates calls to the I/O runtime routines to implement I/O statements, such as `READ` and `WRITE`. The compiler automatically loads runtime routines from `libI77` (on C Series) or `libcl.a` (on SPP Series).

I/O operation

Fortran file I/O to a logical unit consists of:

- Open statement (this can be implicit) (OPEN)
- Series of I/O statements (READ, WRITE, PRINT, ACCEPT, ENCODE, DECODE)
- Optional close statement (CLOSE)

Each I/O transfer (READ, WRITE, PRINT, ACCEPT, ENCODE, DECODE) is implemented as a sequence of operations:

- Initialize I/O transfer
- Series of I/O transmissions
- Terminate transfer

For example, the I/O statement

```
READ (5,100) a, b, c
```

is compiled into the following C Series runtime routine references:

```
for$s_rsfe; start read sequential formatted external
for$do_fio; do formatted I/O for a
for$do_fio; do formatted I/O for b
for$do_fio; do formatted I/O for c
for$e_rsfe; end read sequential formatted external
```

If the I/O list is empty, as in READ (5,100), the for$do_fio calls are not used. But, the end-io-call, for$e_rsfe in this example, is always required.

I/O runtime routine naming convention (C Series)

The I/O transfer type is defined by the following attributes:

Read or write:	r or w
Sequential or direct:	s or d
Formatted, unformatted, list-directed, namelist:	f, u, l, or n
External or internal:	e or i

On CONVEX C Series machines, many of the runtime routine names are constructed using the letters listed above. For

example, the runtime routine `for$s_rsfe` initializes I/O for a read-sequential-formatted-external transfer. The following combinations of I/O types are invalid:

- `ui`—unformatted/internal
- `dl`—direct/list-directed
- `dn`—direct/namelist
- `ni`—namelist/internal

I/O list initialization

These runtime routines prepare a unit for I/O. The following operations are performed:

- If not currently open, opens file with default file attributes, for example, `RECORDTYPE`
- Initializes logical record buffer
- Compiles runtime routine formats
- Saves `ERR` and `END` flags
- Checks for various error conditions, for example, illegal unit number or formatted I/O to file open for unformatted access

Entry points for C Series I/O list initialization are listed below:

```
for$s_{r,w}s{f,u,l,n}e
for$s_{r,w}d{f,u}e
for$s_{r,w}s{f,l}i
for$s_{r,w}dfi
for$s_encode
for$s_decode
```

I/O list element transmission

I/O list element transmission runtime routines perform the actual I/O transmission. There are two levels of buffering in I/O runtime routines: logical record buffering and physical record buffering. The record buffers are filled and flushed, as required by the I/O transmission runtime routines. Each unformatted I/O statement transfers a single logical record; formatted and list-directed I/O statements can transfer multiple records. The physical record size corresponds to the file system block size.

Each call transmits a single value, except for arrays. Arrays are transmitted with a single runtime routine call. If referenced in

column-major order, arrays in implied DO-lists are also transmitted with a single runtime routine.

Formatted transfer of complex values generates two runtime routine calls—one for the real part and one for the complex part. List-directed transfer of complex values generates a single runtime routine reference.

The entry point for C Series I/O list element transmission is as follows:

```
for$do_{f,u,l}io
```

I/O list termination

The following I/O list termination runtime routines complete an I/O operation. This can require flushing the logical record buffer and completion of formatting, if any elements remain in the format string. Different routines are used on C Series machines and SPP Series machines.

C Series I/O list termination runtime routines:

```
for$e_{r,w}s{f,l,u}e
for$e_{r,w}d{f,u}e
for$e_{r,w}s{f,l}i
for$e_{r,w}dfi
for$e_encode
for$e_decode
```

Auxiliary I/O operations

The following auxiliary I/O operation runtime routines implement auxiliary I/O statements. These statements initialize I/O (OPEN), terminate I/O (CLOSE), position files (BACKSPACE, ENDFILE, REWIND), and return information about a file or unit (INQUIRE).

A list of I/O runtime routines and the C Series Fortran statements that they implement follows.

I/O statement	C Series I/O runtime routine
BACKSPACE	for$back
CLOSE	for$close
ENDFILE	for$end
INQUIRE	for$inqu
OPEN	for$open
REWIND	for$rew

IEEE compatibility

F

IEEE compatibility

This appendix describes CONVEX Fortran compatibility with the IEEE 754 standard.

C Series only

CONVEX processors process data encoded in IEEE format but do not perform IEEE standard arithmetic. IEEE format is especially useful for applications that require preprocessing and postprocessing software on graphics workstations. These applications include:

- Finite element codes
- Computational chemistry codes requiring graphics pre- and postprocessing
- Libraries interfacing to workstation applications
- Graphics programs exchanging data with workstations

On CONVEX C Series machines, when the correct processor status word (PSW) bit is set, CONVEX processors produce a binary floating-point format similar to that defined in the *IEEE 754 Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985).

On CONVEX SPP Series machines, floating-point numbers comply with the IEEE standard.

C Series only

CONVEX processors do not support the complete floating-point arithmetic, conversion, and exception operations defined by the standard.

CONVEX processors support basic single- and double-precision floating-point formats, including special numbers such as NaN (Not a Number), Inf (infinity), and signed zero. CONVEX does not support the extended floating-point formats that are optional in the IEEE 754 standard.

C Series only

CONVEX processors use the same algorithms to process native and IEEE format numbers and use a single rounding algorithm. This rounding method is not consistent with the default

rounding method or the directed methods specified by the IEEE standard.

C Series only

Although CONVEX processors can accept special numbers as part of an arithmetic operation, they do not necessarily produce the results specified by the IEEE standard. CONVEX does not support the IEEE specifications for arithmetic operations and conversions in the following areas:

- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings

C Series only

CONVEX processors do not handle exceptions as specified by the IEEE standard, especially in the area of quiet and signaling `NaNs`.

C Series only

If an application is compiled with the IEEE compiler option (`-fi`) on a CONVEX processor, the binary files it produces can be read on another IEEE machine, assuming compatible file and record formats, and identical byte ordering.

System limits

This chapter lists the maximum sizes for finite-length entities in CONVEX Fortran programs. The limits on both C Series machines and SPP Series machines are provided in Table 46.

Table 46 System limits for C Series and SPP Series programs

Finite entity	C Series maximum	SPP Series maximum
Statement length	13,200 characters	13,200 characters
Hollerith length	2000 characters	2000 characters
String length	65,535 characters	65,535 characters
Identifier length	42 characters	42 characters
File name length	200 characters	255 characters
File size	1 terabyte minus 512 bytes*	1 terabyte minus 512 bytes*
`INCLUDE` nesting	127	127
Number of files in a program	127	127

*Other factors, such as the size and number of records in a file, or the NFS protocol, can limit the maximum file size.

Additionally, the total space used by all dimensioned arrays in a program must not exceed the memory space of the system architecture.

ASCII character set

H

Table 47 lists the American Standard Code for Information Interchange (ASCII) with each character equivalent in hexadecimal and octal values. The Fortran character set is a subset of the ASCII character set; although not all ASCII characters are Fortran characters, all of the Fortran characters are included in the ASCII character set.

Table 47 ASCII character set

Hex value	Octal value	Char	Hex value	Octal value	Char	Hex value	Octal value	Char
00	000	NUL	10	020	DLE	20	040	SP
01	001	SOH	11	021	DC1	21	041	!
02	002	STX	12	022	DC2	22	042	"
03	003	ETX	13	023	DC3	23	043	#
04	004	EOT	14	024	DC4	24	044	$
05	005	ENQ	15	025	NAK	25	045	%
06	006	ACK	16	026	SYN	26	046	&
07	007	BEL	17	027	ETB	27	047	'
08	010	BS	18	030	CAN	28	050	(
09	011	HT	19	031	EM	29	051	)
0A	012	LF	1A	032	SUB	2A	052	*
0B	013	VT	1B	033	ESC	2B	053	+
0C	014	FF	1C	034	FS	2C	054	,
0D	015	CR	1D	035	GS	2D	055	-
0E	016	SO	1E	036	RS	2E	056	.

Table 47 (continued) ASCII character set

Hex value	Octal value	Char	Hex value	Octal value	Char	Hex value	Octal value	Char
0F	017	SI	1F	037	US	2F	057	/
30	060	0	50	120	P	70	160	p
31	061	1	51	121	Q	71	161	q
32	062	2	52	122	R	72	162	r
33	063	3	53	123	S	73	163	s
34	064	4	54	124	T	74	164	t
35	065	5	55	125	U	75	165	u
36	066	6	56	126	V	76	166	v
37	067	7	57	127	W	77	167	w
38	070	8	58	130	X	78	170	x
39	071	9	59	131	Y	79	171	y
3A	072	:	5A	132	Z	7A	172	z
3B	073	;	5B	133	[	7B	173	{
3C	074	<	5C	134	\	7C	174	|
3D	075	=	5D	135	]	7D	175	}
3E	076	>	5E	136	^	7E	176	~
3F	077	?	5F	137	_	7F	177	DEL
40	100	@	60	140	`			
41	101	A	61	141	a			
42	102	B	62	142	b			
43	103	C	63	143	c			
44	104	D	64	144	d			
45	105	E	65	145	e			
46	106	F	66	146	f			
47	107	G	67	147	g			
48	110	H	68	150	h			

Table 47 (continued) ASCII character set

Hex value	Octal value	Char	Hex value	Octal value	Char	Hex value	Octal value	Char
49	111	I	69	151	i			
4A	112	J	6A	152	j			
4B	113	K	6B	153	k			
4C	114	L	6C	154	l			
4D	115	M	6D	155	m			
4E	116	N	6E	156	n			
4F	117	O	6F	157	o			

Preprocessor

This appendix describes the CONVEX Fortran preprocessor. Previous versions of the Fortran compiler had a preprocessor. This preprocessor now is optional and is retained for support only. Do not use it unless it is absolutely necessary. If you use it, note the following limitations:

- Macros in Hollerith strings are not expanded and can cause problems with some programs.
- Recursive macros cannot be preprocessed.
- The preprocessor incorrectly handles **CTRL-L** (form feed) even though the compiler treats it (correctly) as a null line.
- Certain language elements may not be available when the preprocessor is used.

Preprocessor statements

Preprocessor statements begin with the # symbol and are syntax-independent of the compiler. Except for the `#define` and `#if` statements, you can continue long statements by entering a backslash (\) at the end of the line to be continued.

`#define` statement

The `#define` statement causes the preprocessor to replace subsequent instances of an *identifier* with a given *token string*. This statement has the following form:

`#define` *identifier token string*

or

`#define` *identifier (identifier,...,) token string*

The *token string* in the definition replaces the *identifier*. The arguments in the call in the second form are *token strings*

separated by commas. Commas within quoted strings or commas protected by parentheses do not separate arguments. The corresponding *token string* from the call replaces every *identifier* mentioned in the formal parameter list of the definition, and the number of formal and actual parameters must be the same. Text inside a character string is not replaced.

Blanks are significant in `#define` statements. The argument list must immediately follow the macro name on the same line; continuations are not allowed.

`#undef` statement

The `#undef` statement causes the identifier preprocessor definition to be removed. This statement has the following form:

`#undef` *identifier*

`#include` statement

The `#include` statement replaces the line on which the statement appears with the contents of a specified file. This statement has the following form:

`#include "`*filename*`"`

This statement searches for the specified file in the directory of the original source file, then along the paths specified via the `-I` option on the command line. `#include` statements can be nested.

`#if` statement

The `#if` statement checks whether a constant expression evaluates to nonzero and whether the identifier is defined or undefined in the preprocessor. This statement has one of the following forms:

`#if` *constant-expression*

or

`#ifdef` *identifier*

or

`#ifdef` *identifier*

These forms are followed by an arbitrary number of lines that can contain a control line `#else`. The last line must be `#endif`. If the condition is true, the lines between the `#else` and the `#endif` are ignored; if the condition is false, the lines between the `#if` and an `#else` (or the `#endif`, if no `#else` exists) are ignored. These constructions can be nested. Continuations of the `#if` statement are not allowed, and you cannot place an `#if` statement within a continued Fortran statement.

When used in a subroutine (or in a main program that is preceded by a subroutine in the same file), `#ifdef` statements cause the preprocessor to generate incorrect line numbers. If you plan to use the CONVEX symbolic debugger (`csd`) on your program, avoid using `#ifdef` statements in any module (other than a main module) that does not contain preceding subroutines in the same file.

Preprocessor options

Several Fortran command line options are associated exclusively with the preprocessor. For a complete list of these options, refer to the "Preprocessor options" section of Chapter 1.

Preprocessor messages

During compilation, the Fortran preprocessor generates self-explanatory error messages or warning messages in the following format:

```
fpp: file: line_number: error message
```

User-defined preprocessors

You can invoke a user-defined preprocessor with the `-pp` command line option. It has the following form:

`-pp=`*name*

where *name* is the name of the preprocessor.

`-pp` must conform to these rules:

- You can supply preprocessor options or text following *name* only if the entire string (*name* and its options) is quoted, as shown in the following example:

  ```
  fc -pp="mypreproc -opt text_stuff" a.f
  ```

- The user-supplied preprocessor is called using the system routine in `libc.a`; therefore, both shell scripts and executable programs are allowed. When the user's

preprocessor is invoked, any user-supplied options or text are passed first, followed by the input file name, then a system-generated output file name of the form "/tmp/pofcl*nnnnnn*.i" where *nnnnnn* is the driver's process number. In the example shown above, the preprocessor would be called as follows:

```
mypreproc -opt text_stuff a.f /tmp/pofclnnnnnn.i
```

Here the preprocessor reads from the file `a.f` and writes to the file `/tmp/pofcl`*nnnnnn*`.i`. Use of the `-fpp` option also affects the names supplied to the user's preprocessor.

- When `-pp` is used in combination with the `-fpp` option, `fpp` executes first, and the output of `fpp` is the input file for the user's preprocessor.
- The preprocessor must supply an exit status, which controls further compilation of the generated output. A zero status allows the driver to continue, calling the actual compiler (`fskel`) using the file `/tmp/pofcl`*nnnnnn*`.i` as input. Any nonzero status prevents the driver from compiling the output file. The preprocessor should generate its own error message (on `stderr`) if a nonzero status is returned because the driver does not issue any error messages.

Sample preprocessor

The following shell script shows a user-defined preprocessor which emulates the `-dc` compiler command line option by replacing all *D*s (and *d*s) that appear in column 1 with blanks.

```
#!/bin/sh
# mypp -- user defined fortran preprocessor
#
# emulates -dc option
#
echo Preprocessor: $0 $*
if [ ${#} -ne 2 ]
then
  echo "mypp requires 2 and only 2 arguments"
  exit 1
fi
sed -e "s/^[Dd]/ /" <$1 >$2
exit 0
```

Index

Symbols

#include preprocessor statement 262
#undef statement 262
$ (dollar sign)
    in cross-reference report 41
%LOC function 80
%REF function 78
%VAL function 79
. (dot)
    in cross-reference report 41
.fil file 10
_ (underscore)
    in cross reference report 41
    in external names 28, 33, 82
    in libU77.a routine names 27
| & redirection operator
    under csh 63

A

a.out 3, 29
-a1 option 19
absolute value 240
ACCEPT statement 66, 68
adb 56
address of variable
    finding with %LOC 80
advisory messages 163
    suppressing 22
-align cseries option 26
-align spp option 26
analyzer
    performance 20
AND bitwise functions 236
ANSI/IEEE standard 754-1985 253
-ansi77 option 8
-ansi90 option 8
-ansic option 27
Application Compiler (APC) 61
arc cosine 228
arc sine 227
arc tangent 228
    degree 229
    two arguments 228
    two arguments (degree) 229
argument packets 17, 75, 86
    and character-valued functions 76
    and non-Fortran programs 78
    built-in functions 78
argument passing
    by reference 78
    by value 79
    code examples 87
    methods of 77
argument pointer 75
arguments 222
    specifying to assembler 28
    specifying to loader 28
    specifying to preprocessor 28
    see also compiler options
arithmetic exceptions 15
arithmetic operators 246
array table 32
arrays
    as arguments 91
    assumed-size 19
    block shared 127
    bounds checking 19
    reversing storage order 146
ASCII character set 257
assembly language
    code examples 88
    debugger 56
    function calls 222
    generating from Fortran 3
    generating from source 18
    interfacing with Fortran 33, 82
associated documents
    Application Compiler User's Guide xx
    CXmetrics User's Guide xx
    Fortran 90 standard xxi
    ISO/IEC 1539:1991 xxi
assumed size arrays
    declaring with last dimension of 1 19
auxiliary I/O operations 252

B

-B option 27

BACKSPACE statement 67
BARRIER directive 125
basic block profiler 59
BEGIN_TASKS directive 125
    attributes for use with 126
binary I/O 65
bitwise functions
    AND 236
    circular shift 239
    clear 238
    complement 237
    extract 237
    OR 236
    set 238
    shift 237
    test 238
    XOR 236
block shared arrays 127
BLOCK_LOOP directive 127
BLOCK_SHARED directive 127
-blockloop option 9
bprof profiler 59
    compiling for 21
-br option 9
BYTE (INTEGER*1) data type
    representation 156

C

C language
    code examples 88
    equivalent variable declarations 84
    interfacing with Fortran 32, 33, 82, 92
    library 220
-c option 15, 30
C Series
    external naming 83
    operating system 99
    system limits 255
-cache option 10
cache size
    specifying 10
call point mismatch errors
    in cross-reference report 50
call stack
    and traceback 118
caller/callee routine cross-reference 51
calling conventions 75, 222
    code example 88
calling utility routines 99
callq instruction 32, 222
-cfc option 8
-cfcwpa option 8
character data
    relational functions 239
character-valued functions
    and argument passing 76
    code example 89
CHDIR utility 99
circular shift bitwise functions 239
clear bitwise functions 238
CLOSE statement 67
code examples
    argument passing 87
    assembly-language equivalents 88
    C language equivalents 88
    calling conventions 88
    character-valued functions 89
    interlanguage programming 92
    procedures 87
code-generation options 15
column-major storage of arrays
    reversing 146
COMMON blocks
    external names 82
    in cross-reference report 47
    loose packing 26
    tight packing 26
common logarithm 225
comparison operators 246
compatibility modes 3
compilation times
    limiting 28
    reducing 22
compiler
    features 1
    finding version of 28
    internal error messages 31, 164
    messages 30, 163, 164
compiler directives
    BARRIER 125
    BEGIN_TASKS 125
    BLOCK_LOOP 127
    BLOCK_SHARED 127
    DO_PRIVATE 132, 134
    END_ORDERED_SECTION 141
    END_TASKS 125
    FAR_SHARED 128
    FAR_SHARED_POINTER 129
    FORCE_PARALLEL 129
    FORCE_PARALLEL_EXT 130
    FORCE_VECTOR 131

format for specifying 123
GATE 131
LOOP_PARALLEL 132
LOOP_PRIVATE 134
MAX_TRIPS 131, 135
NEAR_SHARED 135
NEAR_SHARED_POINTER 136
NEXT_TASK 125
NO_BLOCK_LOOP 136
NO_LOOP_DEPENDENCE 137
NO_PARALLEL 138
NO_PEEL 138
NO_PROMOTE_TEST 138
NO_RECURRENCE 138
NO_SIDE_EFFECTS 139
NO_VECTOR 140
NODE_PRIVATE 136
NODE_PRIVATE_POINTER 137
ORDERED_SECTION 141
PEEL 141
PEEL_ALL 141
PREFER_PARALLEL 141
PREFER_PARALLEL_EXT 144
PREFER_VECTOR 145
PROMOTE_TEST 145
PROMOTE_TEST_ALL 145
PSTRIP 145
ROW_WISE 146
SAVE_LAST 147
SCALAR 148
SELECT 148
SYNCH_PARALLEL 149
TASK_PRIVATE 150
tasking directives 125
THREAD_PRIVATE 151
THREAD_PRIVATE_POINTER 151
UNROLL 152
UNROLL_AND_JAM 152
VSTRIP 153
COMPILER ERROR message 31, 164
compiler options
-72 29
-al 19
-align cseries 26
-align spp 26
-ansi77 8
-ansi90 8
-ansic 27
-B 27
-blockloop 9
-br 9
-c 15, 30
C Series-specific options 7
-cache 10
-cfc 8
-cfcwpa 8
code-generation 15
cross-referencer (fcxref) 36
-cs 19
-cxdb 20, 57
-cxpa 20, 59
-cxpab 20, 59
-cxpalib 20, 59
-cxpamon 21, 59
-cxpar 21
-dc 21
debugging and profiling 19
default options 6
-dfc 8
-ds 10
-ep 10
-errnames 22
-except default 16
-except precise 15
-F66 9, 219
FCOPTIONS environment variable 7
-fi 16, 254
-fn 16
-fpp 25
-I 27
-il 10
-*in* 16
-is 10
-l 219
language compatibility 8
-link 27
-LST 22, 63
-LSTI 22
message and listing 22
-metrics 21
-mi 16
miscellaneous 26
-mo 10
-na 22
-nbr 10
-nga 11
-ngs 11
-nmo 11
-no 11
-noblock 11
-nof90 9
-nopeel 12
-nopm 12
-noptst 12

-nore 17
-nosc 27
-noU77 27, 83
-nsr 12
-nuj 12
-nur 12
-nw 22
-o 28, 29
-O0 13
-O1 13
-O2 13
-O3 13
-On 13
optimization 9
OPTIONS statement 5
-or 22
-p 21
-p8 17
-pb 21
-pcc 28
-pd8 17
-peel 13
-peelall 13
-pg 21
-pl 23
-pp 263
-ppu 28, 84
preprocessor 25, 263
-ptst 14
-ptstall 14
-pw 23
-re 18
-rl 14
-r*n* 18
-S 18
-sa 9
-sc 22
-sfc 9
specifying 4
SPP Series-specific options 7
-sr 14
-tl 28
-tm 18
-uj 14
-ujn 15
-uo 15
-ur 15
-urn 15
-vfc 9
-vn 28
-W 28
-xr 23, 38
-xrl 25
-xra 23, 39
-xro 24
see also Preprocessor options
compiling programs 4
complement bitwise functions 237
COMPLEX data type
calling convention 222
changing default size 17
conjugate 232
imaginary part 231
programmed operators 245
real part 231
composite COMMON block reports 52
conjugate
COMPLEX 232
constants
changing default size 16
default sizes 15
translation of REAL 16
conversion
fix-to-float 231
float-to-fix 230
IEEE/native 239
INTEGER 241
REAL*16 to REAL*4 242
REAL*16 to REAL*8 242
REAL*4 to REAL*16 242
REAL*8 to REAL*16 242
CONVEX Application Compiler 61
CONVEX Consultant
postmortem dump 62
profilers 59
CONVEX Fortran
defined 1
naming conventions 33, 82
CONVEX math library 222
CONVEX visual debugger 57
compiling for 20
ConvexOS 99
system limits 255
utilities 100
see also C Series
cosine 226
hyperbolic 229
COVUEshell 3
Cray Fortran
data type lengths 8
word addresses 8
cross-reference report 40
call point mismatch errors 50
caller/callee routine cross-reference section 51

call-point matching section 48
call-point mismatch error message table 50
COMMON block summary 47
composite COMMON block reports 52
cover page 40
differing members report 53
dollar sign ($) symbol 41
dot (.) symbol 41
exhaustive members list 53
include file notation 42
include file reference table 55
line numbers 42
module interface reports 47
module interfaces section 48
routine reports 41
structure field notation 41
structure summary 46
symbol summary 42
table of contents 55
_ symbol 41
VMS structure notation 41
cross-referencer (fcxref) 23, 35
files and required utilities 56
individual source files 36, 39
multiple source files 39
old 24
output example 43
cross-referencer options 23, 36
-xr 38
-xra 40
-xrm 48
-cs option 19
.cshrc file 6
CXdb 57
capabilities 57
compiling for 20
windows 57
-cxdb option 20, 57
CXmetrics 21
CXpa 58
compiling for 20
-cxpa option 20, 59
-cxpab option 20, 59
-cxpalib option 20, 59
-cxpamon option 21, 59
-cxpar option 21

D

D as debugging statement indicator 21
-D preprocessor option 25
data format 71
data representation 84, 155
BYTE (INTEGER*1) 156
COMPLEX 160
INTEGER 156
LOGICAL 155
REAL 156
two's complement 156
data types
see data representation
default sizes 15
date utility 104
-dc option 21
DCL support 3
debugging
assembly-language 56
compiler options 19
CONVEX CXdb program 57
object code 56
statements 21
DEC Fortran
see VAX Fortran
DECODE statement 66
default compiler options 5
default storage size
changing 16
#define preprocessor statement 261
dependency
parallelizing loops with 149
development tools
Application Compiler 61
Assembly-language debugger 56
cross reference generator 35
CXdb debugger 57
error utility 63
performance analyzer 58
program 35
-dfc option 8
diagnostic messages 30
difference
positive 234
Digital Command Language (DCL) support 3
directives
see compiler directives
DO_PRIVATE directive 132, 134
dollar sign ($)
in cross-reference report 41
dot (.)
in cross-reference report 41
-ds option 10
dump

postmortem 62
dynamic selection 14, 148

E

-E preprocessor option 25
#else preprocessor statement 263
ENCODE statement 67
END specifier 107, 108
END_ORDERED_SECTION directive 141
END_TASKS directive 125
ENDFILE record 69
ENDFILE statement 67, 69
#endif preprocessor statement 263
end-of-file specifier 107
entry points
    I/O list element transmission 251
    I/O list initialization 250
    intrinsics 222
environment variables 6
-ep option 10
ERR specifier 107, 108
-errnames option 22, 164
error 209
error messages 30, 163
    compiler messages 163
    examples of 31
    I/O 107
    redirecting 163
    runtime 31
    runtime I/O (C Series) 209
    runtime I/O (SPP Series) 214
error processing
    I/O 107
    runtime 107
    utilities 113
error utility 63, 164
errsns utility 104
errtrap utility 114
    arguments 115
    hardware differences 115
-except default option 16
-except precise option 15
    overriding 16
exceptions 109, 112
    runtime 107
executable file 29
    specifying name for 28
executing programs 30
Exemplar
    see SPP Series
exit utility 104
exponential functions 225
exponentiation programmed operators 243
external files 70
    accessing 70
    connecting to units 70
    opening 71
external naming
    in CONVEX C 82
    in CONVEX Fortran 82
    -noU77 option 83
    on C Series 83
    on SPP Series 83
    -ppu option 84
extraction bitwise functions 237

F

.f filename extension 3
-F66 option 9, 219
FAR_SHARED directive 128
FAR_SHARED_POINTER directive 129
far-shared memory 128, 129
fc command 4, 29
    default compilation process 2
FCOPTIONS environment variable 4
fcxref program 23, 35
    options 36
    report 40
    see also cross referencer (fcxref)
.fcxrefData file 36
-fi option 16, 254
.fil files
    generating 10
files 70
    .cshrc 6
    executable 29
    external 70
    .fcxrefData 36
    .fil 10
    Fortran source 3
    internal 70
    limits 255
    .met 21
    naming conventions 3
    opening 71
    standard error (stderr) 30
    types of 70
FIND statement 67

fix-to-float conversion 231
floating-point data representation 156, 158
  IEEE 16, 159, 253
  native 157
float-to-fix conversion 230
-fn option 16
footnotes
  in optimization report 32
.FOR filename extension 3
for$ prefix 222
FORCE_PARALLEL directive 129
FORCE_PARALLEL_EXT directive 130
FORCE_VECTOR directive 131
formatted I/O 65
formatted I/O records 69
FOR*nnn*OPEN 71
fort.*nnn* 71
FORTRAN 66 compatibility 9
Fortran 90
  disabling compatibility 9
  intrinsics 243
Fortran argument packets 75
Fortran character set 257
Fortran I/O library 248
Fortran intrinsic library 222
Fortran preprocessor (fpp) 25
fpp 261
-fpp preprocessor option 25
fsplit utility 29
ftn*xxx* 71
function-naming convention 222
functions
  intrinsic 224
  %LOC 80
  %REF 78
  return values 79
  %VAL 79
fxref cross-reference generator 24
  see also cross-referencer (fcxref)

G

GATE directive 131
gerror utility 119
gprof profiler 59
  compiling for 21
graph profiler 59

H

hardware intrinsics 115
hyperbolic cosine 229
hyperbolic sine 229
hyperbolic tangent 229

I

-I compiler option 27
-I preprocessor option 27
I/O
  auxiliary operations 252
  binary 65
  C Series and SPP Series differences 66
  error processing 107
  files 70
  formatted 69
  forms of 65
  invalid type combinations 250
  list-directed 65
  namelist-directed 65
  operations 249
  runtime errors 209
  runtime errors (C Series) 209
  runtime errors (SPP Series) 214
  runtime routine naming convention 249
  status specifier 107
  transfer operations 249
  unformatted 65
I/O list
  element transmission 250
  initialization 250
  termination 251
I/O records 68
  ENDFILE 69
  formatted 69
  in internal files 70
  unformatted 69
I/O statements
  ACCEPT 66
  BACKSPACE 67
  CLOSE 67
  DECODE 66
  ENCODE 67
  ENDFILE 67
  FIND 67
  INQUIRE 67
  OPEN 67

PRINT 66
READ 66
REWIND 67
TYPE 66
WRITE 66
idate utility 104
IEEE compatibility 253
unsupported features 254
IEEE data format 156
floating-point representation 16, 159
range 159
sign 159
IEEE/CONVEX native conversions 239
ierrno utility 119
#if preprocessor statement 262
#ifdef preprocessor statement 263
-il option 10
imaginary part of COMPLEX 231
-in option 16
INCLUDE files
expanding in program listing 22
in cross-reference report 42
reference table 55
index function
string 239
Inf operand 159, 253
inline substitution 10
input/output, *see* I/O
INQUIRE statement 67
INTEGER data type
changing default size 16, 17
conversion 241
function to find nearest 241
representation 156
INTEGER part of REAL 231
INTEGER*1 data type
range 156
representation 156
INTEGER*2 data type
range 156
representation 156
INTEGER*4 data type
range 156
representation 156
INTEGER*8 data type
range 156
representation 156
interlanguage programming
and argument passing 78
code examples 92
Fortran and non-Fortran code 80
subprogram return values 85
internal compiler errors 31, 164
internal files 70
I/O records in 70
interprocedural optimization 61
intrinsic functions 224
intrinsic library 222
invoking the compiler 4
IOSTAT specifier 107, 108
-is option 10
-iw option
to fxref 24

L

-l option 219
language-compatibility options 8
LEN string length function 239
length function
string 239
libc.a 27
libc.a library 220
libF77.a library 222
libI66.a library 219
libI77.a library 248
libraries
C Series libraries 220
Fortran I/O 248
intrinsic 222
libc.a 220
libF77.a 222
libI66.a 219
libI77.a 248
libU77.a 27, 219
math 222
runtime 219
SPP Series libraries 221
libU77.a library 27, 219
limits
file size 70, 255
Fortran 255
system 255
line numbers
in cross-reference report 42
-link options 27
list-directed I/O 65
listing options 22
loader 29
functions 29
passing options via fc 27
loading programs 29

suppressing 15
%LOC function 80
logarithm
common 225
natural 225
LOGICAL data type
changing default size 16, 17
representation 155
logical name 71
logical record buffering 250
longjmp utility 113
loop blocking 9
preventing 11, 136
loop peeling 13, 141
preventing 12, 138
loop replication 14
loop table 32
loop unrolling 14, 15
LOOP_PARALLEL directive 132
LOOP_PRIVATE directive 134
-LST option 22, 63, 164
-LSTI option 22

M

machine-independent optimizations 13
man pages xxi
math library 222
MAX_TRIPS directive 131, 135
maximum intrinsic functions 232
maximum strip mine lengths 153
memory
block-shared 127
far-shared 128, 129
near-shared 135
node-private 136
thread-private 151
messages 30
compiler messages 163
options 22
see also error messages
.met file 21
metrics 21
-metrics option 21
-mi option 16
minimum intrinsic functions 233
miscellaneous compiler options 26
-mo option 10
module interfaces
in cross-reference report 47
mth$ prefix 222
mvbits utility 105

N

-na option 22
naming conventions 82
C Series and SPP Series differences 33
C Series runtime routines 222
NaN operand 159, 253
native floating-point representation 16, 157
natural logarithm 225
-nbr option 10
NEAR_SHARED directive 135
NEAR_SHARED_POINTER directive 136
nearest integer intrinsics 241
near-shared memory 135
NEXT_TASK directive 125
-nga option 11
-ngs option 11
-nmo option 11
-no option 11
NO_BLOCK_LOOP directive 136
NO_LOOP_DEPENDENCE directive 137
NO_PARALLEL directive 138
NO_PEEL directive 138
NO_PROMOTE_TEST directive 138
NO_RECURRENCE directive 138
NO_SIDE_EFFECTS directive 139
NO_VECTOR directive 140
-noblock option 11
NODE_PRIVATE directive 136
NODE_PRIVATE_POINTER directive 137
node-private memory 136
-nof90 option 9
non-Fortran to Fortran calling sequence 80
-nopeel option 12
-nopm option 12
-noptst option 12
-nore option 17
-nosc option 27
notational conventions xviii
-noU77 option 27, 83
-nsr option 12
-nuj option 12
-nur option 12
-nw option 22

O

-O option 13
-o option 28, 29
-O1 option 13
-O2 option 13
-O3 option 13
old cross-referencer 24
OPEN statement 67, 71
operating system
  C Series 99
  ConvexOS 99
  Fortran routines 100
  SPP Series 99
  SPP-UX 99
  system limits 255
operators
  COMPLEX 245
  exponentiation 243
  REAL*16 246
  string-manipulation 247
  vector mask 247
optimization
  disabling 11
  machine independent 13
  potentially unsafe 15
optimization options 9
  -blockloop 9
  -br 9
  -cache 10
  -ds 10
  -ep 10
  for optimization report 32
  -il 10
  -is 10
  -mo 10
  -nbr 10
  -nga 11
  -ngs 11
  -nmo 11
  -no 11
  -noautopar 11
  -noautovec 11
  -noblock 11
  -nopeel 12
  -nopm 12
  -noptst 12
  -nsr 12
  -nuj 12
  -nur 12
  -O0 13
  -O1 13
  -O2 13
  -O3 13
  -or 22, 32
  -peel 13
  -peelall 13
  -ptst 14
  -ptstall 14
  -rl 14
  -sr 14
  -uj 14
  -ujn 15
  -uo 15
  -ur 15
  -urn 15
optimization report 22, 31
  options for 32
  tables in 32
  variable name footnotes 32
options
  see compiler options
OPTIONS statement 4, 5
OR bitwise functions 236
-or option 22
ORDERED_SECTION directive 141

P

-p option 21, 60
-p8 option 17
packets
  see argument packets
parallelization
  forcing in loops 129
  preventing 148
  preventing in loops 138
  strip mining 145
  synchronizing 149
passing arguments to subprograms 75
pattern matching
  preventing 12
-pb option 21, 60
-pcc option 28
-pd8 option 17
PEEL directive 141
-peel option 13
PEEL_ALL directive 141
-peelall option 13
peeling
  see loop peeling

performance analyzer 58
compiling for 20
perror utility 119
-pg option 21, 60
physical record buffering 250
-pl option 23
pmd utility 62
pointer
argument 75
compiler-generated 129, 136, 151
far-shared pointer 129
near-shared 136
thread-private 151
portable C compiler
mixing code with Fortran programs 28
positive difference 234
postmortem dump 62
-pp option 263
-pp preprocessor option 26
-ppu option 28, 84
PREFER_PARALLEL directive 141
PREFER_PARALLEL_EXT directive 144
PREFER_VECTOR directive 145
preprocessor 261
#define statement 261
#else statement 263
#endif statement 263
example user-defined 264
Fortran 25
#if statement 262
#ifdef statement 263
#include statement 262
messages 263
statements 261
#undef statement 262
user-defined 263
preprocessor options 25, 263
-D 25
-E 25
-fpp 25
-I 27
-pp 26
-U 26
PRINT statement 66, 68
private variables
DO_PRIVATE 132, 134
TASK_PRIVATE 150
procedures
code example 87
processors
specifying expected number 10
prof profiler 59
compiling for 21
profilers 59
bprof 21
CXpa 20
gprof 21
prof 21
profiling options 19
program development tools 35
program interfaces 32
program listings
generating 22
PROMOTE_TEST directive 145
PROMOTE_TEST_ALL directive 145
PSTRIP directive 145
-ptst option 14
-ptstall option 14
-pw cross-reference option 24
-pw option 23

R

ran utility 105
-re option 18
READ statement 66, 68
REAL constants
IEEE translation 16
translation 16
REAL data type
as part of COMPLEX 231
changing default size 17, 18
INTEGER part 231
representation 156
REAL*16 data type 222
range 157
representation 156
REAL*16 programmed operators 246
REAL*16 to REAL*4 conversion 242
REAL*16 to REAL*8 conversion 242
REAL*4 data type representation 156
REAL*4 to REAL*16 conversion 242
REAL*4 to REAL*8 conversion 242
REAL*8 data type representation 156
REAL*8 product of REAL*4 233
REAL*8 to REAL*16 conversion 242
REAL*8 to REAL*4 conversion 242
records
ENDFILE 69
formatted I/O 69
I/O 68
in internal files 70

logical buffering 250
physical buffering 250
unformatted I/O 69
recurrence
disregarding 138
redirecting stderr under csh 63
reentrant code
generating on C Series 18
suppressing on SPP Series 17
%REF function 78
registers
scalar 222
vector 222
relational functions
character 239
remainder 235
replication
loop 14
report
optimization 22, 31
reserved operand 157
return values 85
REWIND statement 67
-rl option 14
-r*n* option 18
Rop operand 157
ROW_WISE directive 146
cautions 146
row-major storage of arrays
forcing 146
rtnq instruction 222
runtime errors 107
messages 31
runtime exceptions 107, 112
runtime interface 32
runtime libraries 219
runtime routines
data items 248
names 222
prefixes 222
utilities 99
runtime stack 80

S

.s filename extension 3
-S option 18
-sa option 9
SAVE_LAST directive 147
-sc option 22
SCALAR directive 148
scalar loop execution
forcing 148
scalar registers 222
scalar replacement 12
scalar truncation 222
secnds utility 105
SELECT directive 148
set bitwise functions 238
setjmp utility 113
-72 option 29
-sfc option 9
shell variables 71
shift
bitwise circular 239
shift bitwise functions 237
sign
of floating-point data 158
transfer of 236
signal handling
examples 120
signal utility 118
signals 109
trapped by errtrap 114
sine 226
hyperbolic 229
-sl option 24
software metrics 21
source files 3
specifiers
END 107, 108
ERR 107, 108
IOSTAT 107, 108
SPP Series
external naming 83
operating system 99
system limits 255
SPP-UX 99
system limits 255
-sr option 14
stack, runtime 80
standard error file
see stderr file
standard output
see stdout file
standard profiler 59
statements
ACCEPT 66
BACKSPACE 67
CLOSE 67
DECODE 66
ENCODE 67

ENDFILE 67, 69
FIND 67
INQUIRE 67
OPEN 67, 71
OPTIONS 5
PRINT 66
READ 66
REWIND 67
TYPE 66
WRITE 66
stderr file 30, 72
C Series and SPP Series differences 30
redirecting under csh 63
stdin file 72
stdout file 72
storage element
finding address 80
storage size
changing default 17, 18
string-manipulation programmed operators 247
strings
index function 239
strip mining
maximum strip mine lengths 153
parallel 145, 146
vector 153
subprograms
argument packets 75
argument-passing mechanisms 77
calling conventions 75
external names 82
passing arguments to 75
return values 85
substitute compiler 27
Sun Fortran 9
suppressing warning messages 22
symbol summary
in cross-reference report 42
symbolic assembly code
generating from source 18
SYNCH_PARALLEL directive 149
synchronizing parallel loop execution 149
syntax check compiler option 22
system errors
retrieving message numbers 119
system limits 255
system utilities 99
SYSTEM utility 103
system-detected errors 209

T

table
analysis 32
array 32
loop 32
privatization 32
test 32
variable name footnote 32
tangent 227
hyperbolic 229
target machine for compilation 18
TASK_PRIVATE directive 150
tasking directives 125
attributes for use with 126
test bitwise functions 238
test promotion 14, 145
preventing 12, 138
test table 32
THREAD_PRIVATE directive 151
THREAD_PRIVATE_POINTER directive 151
thread-private memory 151
time limit for compilation 28
time utility 105
-tl option 28
-tm option 18
tools for developing programs 35
traceback utility 118
transfer of sign 236
traper utility
and floating point exceptions 115
two's complement representation 156
TYPE statement 66, 68

U

-U preprocessor option 26
-uj option 14
-ujn option 15
#undef preprocessor statement 262
underscore(_)
in cross-reference report 41
unformatted I/O 65
unformatted I/O records 69
header 69
trailer 69
units 70
connecting to external files 70
implicit unit numbers 71

redirection 72
unroll and jam transformation 14, 15, 152
UNROLL directive 152
UNROLL_AND_JAM directive 152
unrolling loops 15
unsafe optimizations 15
-uo option 15
-ur option 15
-urn option 15
user-defined preprocessor 263
example of 264
utilities 99
calling 99
CHDIR 99
ConvexOS 100
DATE 104
error 63
error-processing 113
ERRSNS 104
errtrap 114
EXIT 104
gerror 119
IDATE 104
ierrno 119
longjmp 113
MVBITS 105
perror 119
RAN 105
SECNDS 105
setjmp 113
signal 118
SYSTEM 103
TIME 105
traceback 118
VAX 104

V

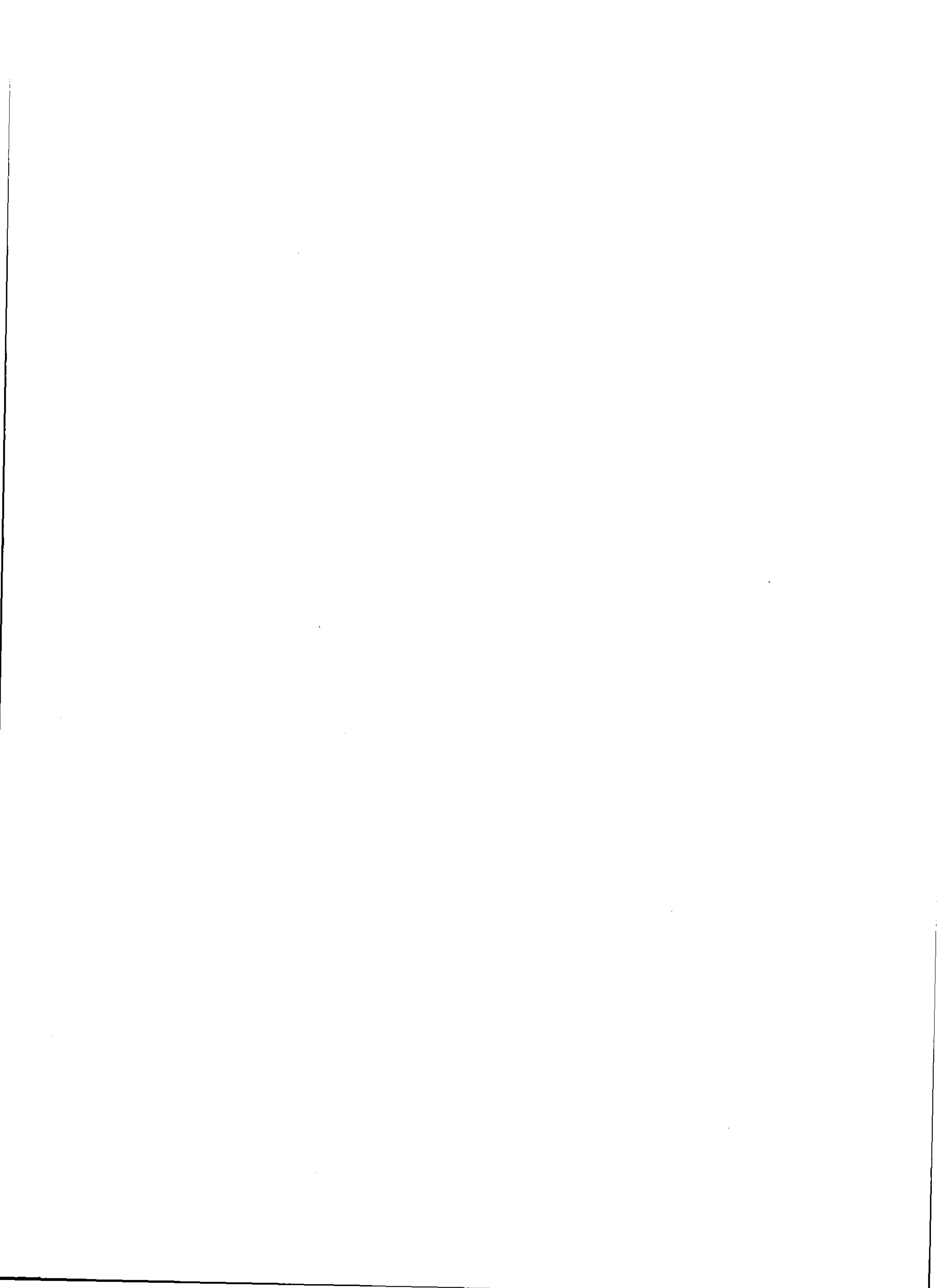
%VAL function 79
values
COMPLEX 222
passing arguments by 79
see also data representation
variable declarations
C correspondence to Fortran 84
variables
changing default size 16
private 132, 134, 150
representation, *see* data representation
variables and constants
changing default size 17, 18
VAX Fortran 9
records in I/O statements 8
VAX-11 utilities 104
vector mask programmed operators 247
vector registers 222
vectorization
forcing in loops 131
preventing 148
preventing in loops 140
version of compiler, finding 28
-vfc option 9
VMS
interface support 3
-vn option 28
VSTRIP directive 153

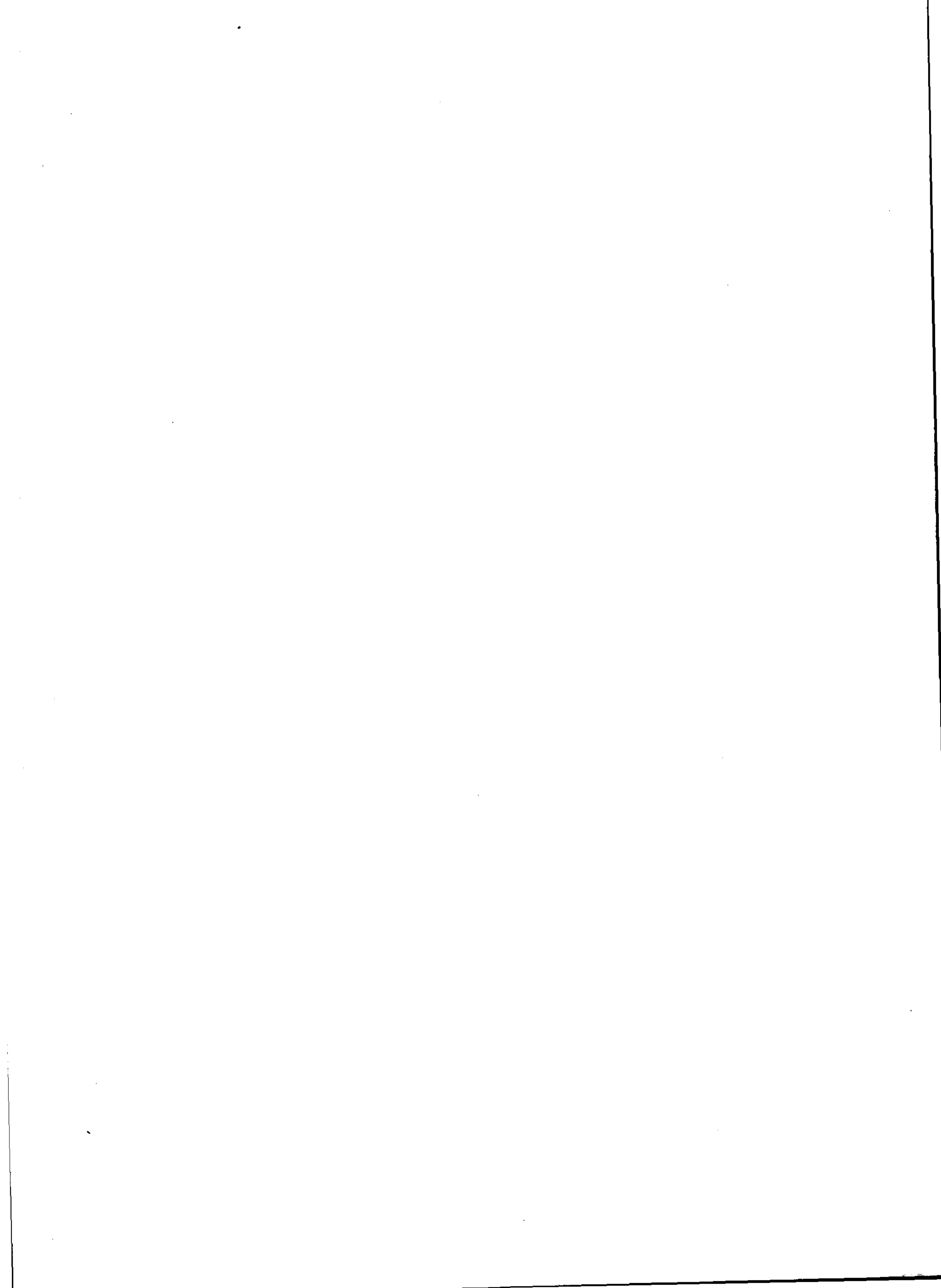
W

-W option 28
warning messages 163
suppressing 22
word addresses
for Cray pointers 8
WRITE statement 66, 68

X

XOR bitwise functions 236
-xr option 23, 38
-xr1 option 25
-xra option 23, 39
in makefiles 40
-xrm option 48
-xro option 24





CONVEX PRESS

ORDER NUMBER
DSW-038

DOCUMENT NUMBER
720-000030-215