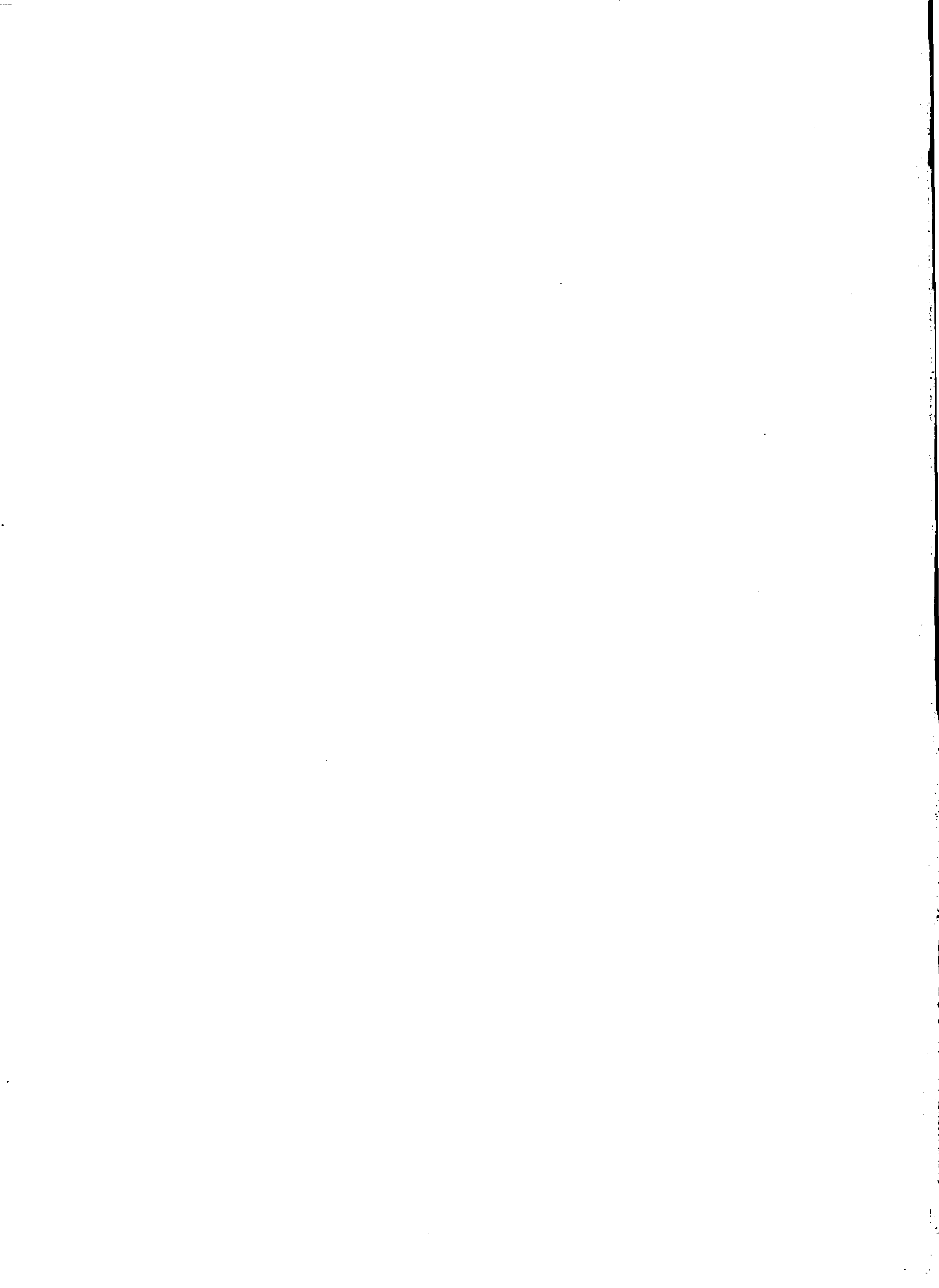
CONVEX FORTRAN Guide

First Edition

CONVEX

CONVEX COMPUTER CORPORATION

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



CONVEX FORTRAN Guide

Order No. DSW-038

First Edition
October 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX FORTRAN
Guide Order No. DSW-038

Copyright 1991 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 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.

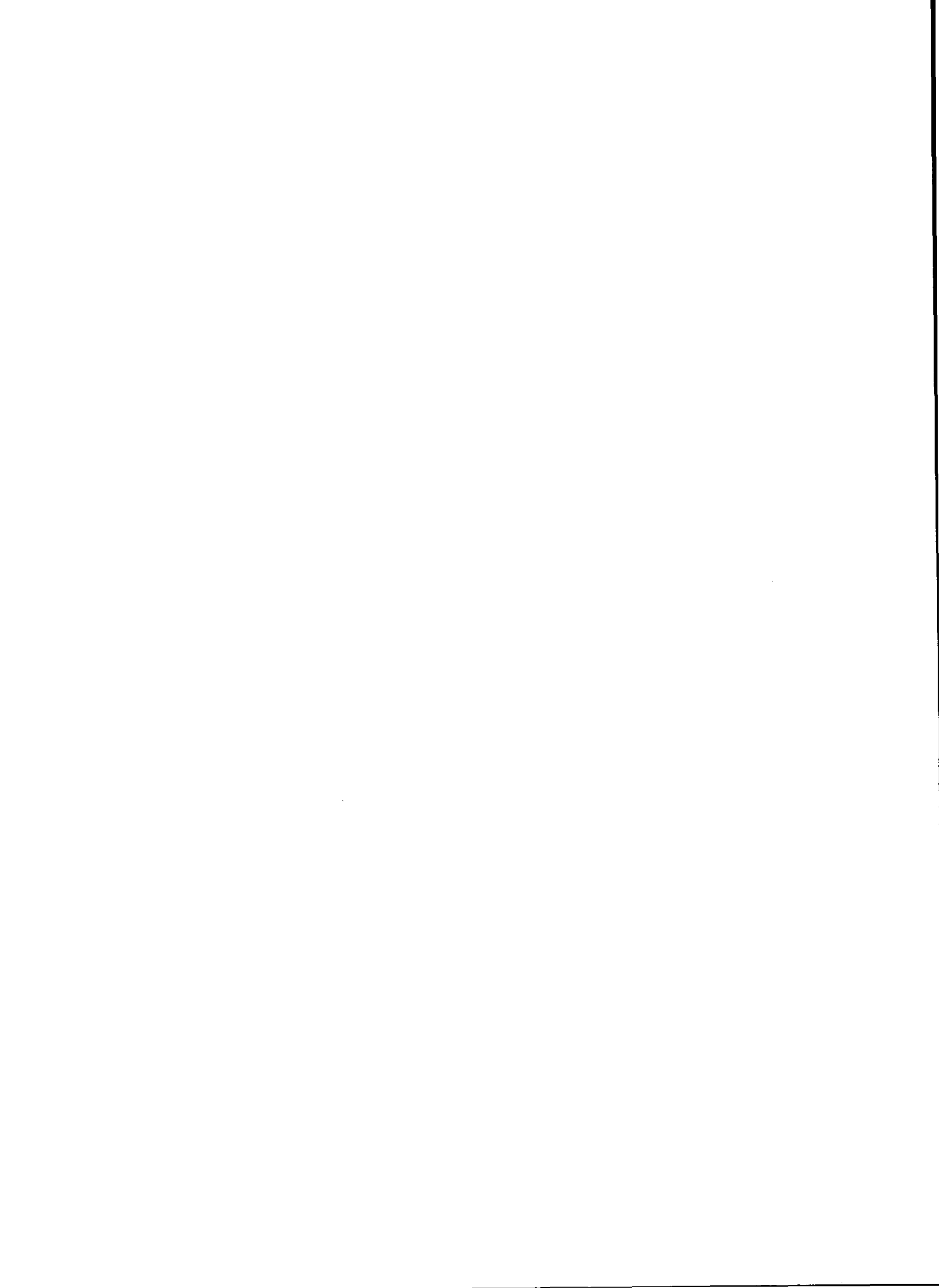
CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
CONVEX C100 Series, C200 Series, C3 Series, C3200 Series, C3400 Series, C3400J Series and C3800 Series are trademarks of CONVEX Computer Corporation.
C1, C120, C210, C220, C230, C240, C3210, C3220, C3230, C3240, C3410, C3420, C3430, C3440, C3460, C3480, C3820, C3840, C3860 and C3880 are trademarks of CONVEX Computer Corporation.
ConvexOS, CXwindows, CXdb, CXpa and CONVEX Consultant are trademarks of CONVEX Computer Corporation.
COVUE is a registered trademark of CONVEX Computer Corporation. COVUE consists of the following products: COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.
Cray is a registered trademark of Cray Research, Inc.
Sun FORTRAN is a trademark of Sun Microsystems, Inc.
UNIX is a trademark of UNIX System Laboratories, Inc.
VAX and VMS are trademarks of Digital Equipment Corporation.

Printed in the United States of America

Revision Information for

CONVEX FORTRAN Guide

Edition	Document No.	Description
First	720-000130-101	Released October 1991. Published *CONVEX FORTRAN User's Guide*, *Language Reference Manual*, and Sections 1F and 3F of the man pages under one cover. Added `fcxref` description to Chapter 4 of the User's Guide. Added `IF-DO` interchange options, `-f90` option, `-nosc` option and various other revisions to Chapter 1 of the User's Guide. Added Fortran 90 array support information to Chapters 2 and 5 and Appendix A of the Language Reference Manual. Added short-circuit evaluation of conditionals section to Chapter 6 of the Language Reference Manual.



Contents

Part 1 CONVEX FORTRAN User's Guide 1

How to use this guide xxv

Purpose and audience xxv
Organization xxv
Scope xxviii
Notational conventions xxviii
Notes xxix
Associated documents xxix
Technical assistance xxxi
The `contact` utility xxxi

1 Compiling programs 3

Overview 3
File-naming conventions 4
Compiling programs 5
  Language-compatibility options 5
  Optimization options 6
  Code-generation options 9
  Debugging and profiling options 12
  Message and listing options 14
  Preprocessor options 16
  Miscellaneous options 17
Loading programs 19
  Using the `fc` command 19
Executing programs 19
Messages 20
  Compiler messages 20
  Optimization report 20
    Loop table, part 1 20
    Loop table, part 2 22
    Array table 22
  Runtime error messages 23
Program interfaces 24
Optimization 25
  Machine-dependent optimization 26
  Local scalar optimization 26
  Global scalar optimization 27
  Vectorization 27

Parallelization 27
Inline substitution 28
Loop replication 28
IF-DO optimizations 29

2 Input/output operations 31

Units 31
Logical names 32
The OPEN statement 33
Assigning logical names 34
Examples 34
Forms of input/output 36
File type 36
Access modes 37
Sequential 37
Direct 37
Logical records 38
Direct-access external file 38
Sequential-access external file 38
Namelist-directed input/output 38
Internal files 38
Data file formats 39
Accessing file pointers 39
Input/output statement summary 41

3 Character data .43

Character constants 43
Declaring character variables 44
Initializing character variables 45
Character substrings 45
Concatenating character strings 46
Character input/output 47
Character library functions 47
ICHAR 47
CHAR 48
LEN and LNBLNK 48
INDEX and RINDEX 49
Lexical comparison functions 49

4 Program development tools 51

Cross-reference generator 51
Cross-referencer options 52
Cross-referencing with -xr 54
Cross-referencing with -xra 54
Cross-reference report 56

Cover page 56
Routine reports 56
Caller/callee routine cross-reference 61
Composite common block reports 62
Include file reference table 64
Table of contents 65
Files 66
`fxref` support 66
Assembly-language debugger 66
Performance analyzer 67
Routine-level profiling 67
Loop-level profiling 68
Block-level profiling 68
CONVEX Consultant 69
Symbolic debugger 69
Postmortem dump 70
Profilers 70
CXdb debugger 71
`error` utility 73

5 Calling conventions 75

FORTRAN subprogram calling convention 75
FORTRAN argument packets 75
Argument-passing mechanisms 77
Argument packet built-in functions 77
%VAL function 77
%REF function 78
Function return values 78
%LOC function 79
Non-FORTRAN-to-FORTRAN calling sequence 79
Procedure names 80
Data representations 81
Return values 81
Argument packets 83
Examples 83
Example 1 83
Example 2 84
Example 3 85
Example 4 86
Example 5 86
Example 6 87

6 System utilities 89

How to call utility routines 89
ConvexOS utilities 90
The `system` utility 92

VAX-11 FORTRAN system utilities 93
date 93
idate 93
errsns 94
exit 94
secnds 94
time 94
ran 94
mvbits 95

7 Runtime errors and exceptions 97

I/O error processing 97
ERR and END specifiers 98
IOSTAT specifier 98
Signals and exceptions 99
Signals 99
Exceptions 101
Error-processing utilities 102
setjmp and longjmp 102
errtrap 102
Hardware differences 104
signal 106
traceback 106
The perror, gerror, and ierrno utilities 107
Examples of signal handling 107

A FORTRAN data representations 111

Logical representation 111
Integer representation 112
Real data representation 112
Native floating-point 114
IEEE floating-point 115
Complex representation 116
Character representation 116
Hollerith representation 116

B Compiler and runtime messages 117

Compiler messages 117
Runtime error messages 118
System errors 118
I/O errors generated by runtime library 118

C Runtime libraries 123

Intrinsic library and math library 125
Calling conventions 125
Function-naming convention 125
Intrinsic runtime routines 127
Exponentiation programmed operators 143
Complex programmed operators 144
REAL*16 programmed operators 144
Vector mask programmed operators 145
String-manipulation programmed operators 146
Runtime routine data items 146
FORTRAN I/O library 147
I/O operation 147
I/O runtime routine naming convention 148
I/O list initialization 148
I/O list element transmission 149
I/O list termination 149
Auxiliary I/O operations 150

D IEEE Compatibility 151

Part 2 CONVEX FORTRAN Language Reference Manual . 153

1 Introduction . . . 155

Types of programs . . . 155
FORTRAN character set . . . 155
Comment line . . . 156
FORTRAN statements . . . 156
Character-per-column formatting . . . 158
Statement label field . . . 158
Initial line . . . 159
Continuation line . . . 159
Statement text field . . . 159
Debug statements . . . 159
Compiler directives . . . 160
ANSI-standard formatting . . . 160
Tab-key formatting . . . 160
Order of statements and lines . . . 160
`OPTIONS` statement . . . 161
`INCLUDE` statement . . . 163

2 FORTRAN statement components . . . 165

Symbolic names . . . 165
Data types . . . 165
Conversion of data types . . . 167
Constants . . . 168
Integer constants . . . 169
Real constants . . . 169
Complex constants . . . 170
Octal constants . . . 170
Hexadecimal constants . . . 171
Hollerith constants . . . 173
Logical constants . . . 174
Character constants . . . 174
Variables . . . 175
Arrays . . . 175
Array declaration . . . 176
Automatic arrays . . . 178
Allocatable array declarations . . . 179
Array subscripts . . . 181
Fortran 90 allocatable arrays . . . 181
Fortran 90 array sections . . . 182
Array storage . . . 184
Fortran 90 array manipulation intrinsics . . . 184
Vector and matrix multiply functions . . . 184
Reduction functions . . . 185
Construction and manipulation functions . . . 190

Location functions 193
Expressions 194
Arithmetic expressions 194
Operator precedence 195
Data type priority 195
Relational expressions 196
Logical expressions 197
Character expressions 198
Character substrings 198
Constant expressions 199

3 Specification statements 201

COMMON statement 202
IMPLICIT statement 203
Type-declaration statements 204
Numeric type-declaration statements 204
CHARACTER type-declaration statements 205
DIMENSION statement 206
EQUIVALENCE statement 206
Equivalencing arrays 207
Equivalencing substrings 209
Using EQUIVALENCE in common blocks 209
PARAMETER statement 210
Standard PARAMETER statement 210
Alternate PARAMETER statement 210
PROGRAM statement 211
NAMELIST statement 212
EXTERNAL statement 213
INTRINSIC statement 213
SAVE statement 214
ALLOCATABLE statement 215

4 DATA statement 217

DATA statement form 217
Implied-DO 219
DATA statement extensions 220

5 Assignment statements 223

Character conversions 224
Fortran 90 array assignments 224
Data conversion rules 225
ASSIGN statement 227

6 Control statements 229

GOTO statements 229
Unconditional GOTO statement 229
Computed GOTO statement 230
Assigned GOTO statement 231
IF statements 232
Arithmetic IF statement 232
Logical IF statement 233
Block IF statement 234
Nested block IF statements 237
Short-circuit evaluation of conditionals 237
DO statement 238
Nested DO loops 240
Extended-range DO loops 240
DO WHILE statement 241
END DO statement 242
CONTINUE statement 243
CALL statement 243
RETURN statement 243
STOP statement 243
PAUSE statement 244
END statement 244

7 Input/output statements 245

Records 246
Formatted records 246
Unformatted records 246
ENDFILE record 247
Files 247
Internal files 247
Units 247
Accessing files 249
Sequential access 249
Direct access 249
I/O statement format 250
Input/Output lists 250
Implied-DO lists 251
Specifiers 252
Unit specifier 252
Format specifier 252
Record specifier 253
Status specifier 254
Error specifier 254
End-of-file specifier 254
Namelist specifier 255
READ statement 255
External sequential READ statements 256

Formatted 256
Unformatted 257
List-directed 257
Namelist-directed 258
External direct READ statements 258
Formatted 259
Internal READ statements 259
Sequential access 260
Direct-access 260
ACCEPT statement 260
WRITE statement 261
Sequential-access WRITE statements 262
Formatted 262
Unformatted 263
List-directed 263
Namelist-directed 263
External direct-access WRITE statements 264
Formatted 264
Unformatted 264
Internal WRITE statements 265
Sequential access 265
Direct access 265
PRINT and TYPE statements 266
Additional statements 266
ENCODE statement 266
DECODE statement 268
FIND statement 270
Auxiliary input/output statements 271
OPEN statement 271
ACCESS keyword 273
ASSOCIATEVARIABLE keyword 274
BLANK keyword 274
BLOCKSIZE keyword 274
CARRIAGECONTROL keyword 275
DEFAULTFILE keyword 276
DISPOSE keyword 277
ERR keyword 277
FILE keyword 277
FORM keyword 278
IOSTAT keyword 279
MAXREC keyword 279
NOSPANBLOCKS keyword 280
READONLY keyword 280
RECL keyword 280
RECORDTYPE keyword 280
STATUS keyword 281
UNIT keyword 282
CLOSE statement 282
INQUIRE statement 283

File-positioning statements 286
REWIND statement 287
BACKSPACE statement 287
ENDFILE statement 288
Binary data file format conversions 288
When to use the conversion feature 289
-dfc option 289
Conversion using OPEN statement 289
Restrictions on conversions 290
Error handling using data format conversions 291
User-defined conversions 292
Sample conversion routine 293
User-suppliedconversion routine names 294
Conversion using a shell variable 296

8 Format specifications 299

FORMAT statement 299
FORMAT control 301
Repeat count 303
Descriptors 303
A descriptor 303
Apostrophe descriptor 305
Asterisk descriptor 305
H descriptor 306
L descriptor 306
I descriptor 307
O descriptor 308
Z descriptor 309
F descriptor 310
E and D descriptors 312
G descriptor 314
B descriptors 316
P descriptor 318
S descriptors 319
R descriptor 320
X descriptor 321
T descriptors 321
$ descriptor 323
Q descriptor 323
Colon descriptor 324
Slash descriptor 324
Default field descriptor values 325
Comma field separator on input data 326
Runtime formats 326
Variable formats 327
List-directed formatting 328
Input 329
Character input 329

Nulls and slashes 330
Namelist-directed formatting 330
List-directed output 333
Namelist-directed output 334
Carriage-control characters 335

9 Subprograms 337

Dummy and actual arguments 337
Variables as dummy arguments 338
Arrays as dummy arguments 339
Adjustable arrays 339
Assumed-size arrays 340
Character arguments 341
Character argument lengths 341
Procedures as dummy arguments 343
Alternate return arguments 343
Functions 344
Intrinsic functions 344
Built-in functions 345
%REF and %VAL functions 345
%LOC function 346
Statement functions 347
Function subprograms 348
Subroutine subprograms 350
ENTRY statement 352
RETURN Statement 353

10 Block data subprogram 357

A Intrinsics and commonly used library routines 359

Generic and specific intrinsics 359
Notes: 371
Commonly used library routines 374

B Preprocessor 377

Preprocessor statements 377
#define statement 377
#undef statement 378
#include statement 378
#if statement 378
Preprocessor options 379
Preprocessor messages 380
User defined preprocessors 380
Sample preprocessor 381

C Compiler directives 383

ASSIGN_LOCK, FREE_LOCK 386
BEGIN_ORDER, END_ORDER 387
BEGIN_SECTION, END_SECTION 387
BEGIN_TASKS, NEXT_TASK, END_TASKS 388
FORCE_PARALLEL_EXT 389
FORCE_PARALLEL 390
FORCE_VECTOR 390
MAX_TRIPS 391
NO_PARALLEL 391
NO_PEEL 391
NO_PROMOTE_TEST 391
NO_RECURRENCE 392
NO_SIDE_EFFECTS 392
NO_VECTOR 393
PEEL 393
PEEL_ALL 393
PREFER_PARALLEL_EXT 394
PREFER_PARALLEL 394
PREFER_VECTOR 394
PROMOTE_TEST 394
PROMOTE_TEST_ALL 394
PSTRIP 395
ROW_WISE 395
SCALAR 396
SELECT 397
SYNCH_PARALLEL 398
UNROLL 399
VSTRIP 399

D System limits 401

E ASCII character set 403

F FORTRAN 66 compatibility 405

Compiling FORTRAN 66 programs 405
EXTERNAL statement 406
DO loop minimum iteration count 406
OPEN statement keywords 407
  BLANK keyword 407
  STATUS keyword 408
X descriptor 408
  Format code separators 408

G Cray FORTRAN compatibility 409

Compiler defaults 409
Unsupported Cray features 410
Cray POINTER support 410
Debugging code containing Cray pointers 411
Cray automatic arrays 411
Cray BUFFERIN, BUFFEROUT support 412
Related statements and routines 412
Restrictions 413
Cray unformatted file support 413
Supported Cray library routines 414
Supported Cray intrinsics 414
Cray TASK COMMON support 415
Cray Boolean octal constant support 415
Cray Hollerith constants 416

H VAX FORTRAN compatibility 417

Supported features 417
Unsupported features 418
Miscellaneous differences 420
VAX FORTRAN records 420
Structure declaration 421
Field declaration 422

I Sun FORTRAN compatibility 425

Part 3 CONVEX FORTRAN Man Pages 429

Section 1F 431

Section 3F 463

Figures

Figure 1 Cross-referencer report cover page 56

Figure 2 Cross-referencer routine report.................................... 59

Figure 3 Cross-referencer routine report structure summary 61

Figure 4 Caller/callee routine cross reference example 62

Figure 5 Cross-referencer common block summary example 64

Figure 7 Cross-referencer table of contents example 65

Figure 6 Include file reference table example............................. 65

Figure 8 Argument packet: example 1 76

Figure 9 Argument Packet: example 2 77

Figure 10 Calling a FORTRAN subroutine 80

Figure 11 C1 series intrinsic errors 105

Figure 12 C2 and C3 series intrinsic errors 105

Figure 13 Traceback resulting from an exception 107

Figure 14 `traceback` as a user-called subroutine 107

Figure 15 Signal handler for interrupts 108

Figure 16 Signal handler for arithmetic exceptions 109

Figure 17 `LOGICAL` data type representation 111

Figure 18 `INTEGER` data type representation 112

Figure 19 `REAL` data type representation 113

Figure 20 `COMPLEX` data type representation 116

Figure 21 CHARACTER data type representation 116

Figure 22 Required order of statements 161

Figure 23 FORTRAN example conversion routine 293

Figure 24 C example conversion routine 294

Tables

Table 1 Optimization levels 26

Table 2 Implicit units 31

Table 3 Examples of default logical names 32

Table 4 Input/output statements 41

Table 5 Lexical intrinsic functions 50

Table 6 Cross-referencer options 53

Table 7 Context operators 58

Table 8 Postmortem dump contents 70

Table 9 Compiler options for profiling 71

Table 10 Built-in functions and defaults for argument lists 78

Table 11 Function return values 79

Table 12 FORTRAN and C declarations 81

Table 13 Complex function: C equivalent 82

Table 14 Character functions: C equivalent 82

Table 15 Character arguments: C equivalent 83

Table 16 Calling Sequences for ConvexOS Utilities 90

Table 17 Signal names and numbers 100

Table 18 Mapping exceptions to signals and codes 101

Table 19 Intrinsic instructions—
C200, C3200, C3400, and C3800 Series 104

Table 20 FORTRAN runtime libraries 124

Table 21 Argument/result codes 126

Table 22 Intrinsic functions 127
Table 23 Exponentiation routines 143
Table 24 Complex programmed operators 144
Table 25 REAL*16 programmed operators 145
Table 26 FORTRAN fields 158
Table 27 OPTIONS statement 162
Table 28 Data types 166
Table 29 Storage requirements for data types 167
Table 30 Arithmetic operators 194
Table 31 Arithmetic operator precedence 195
Table 32 Data type priority 196
Table 33 Logical operator precedence 197
Table 34 Array locations 208
Table 35 Conversion of expressions 225
Table 36 Data transfer I/O statements 245
Table 37 OPEN statement keywords 272
Table 38 INQUIRE specifiers 285
Table 39 Data format conversion routine names 288
Table 40 User-supplied conversion routine names 295
Table 41 Shell variable attributes 296
Table 42 Character assignment for numeric I/O list elements 304
Table 43 Data conversion based on magnitude 315
Table 44 BLANK specifier defaults 317
Table 45 Default field descriptors 325

Table 46 List-directed output formats 333

Table 47 Vertical format control 335

Table 48 Built-in functions and defaults for argument lists ... 346

Table 49 Generic and specific intrinsics 359

Table 50 Restrictions on directive use 385

Table 51 Maximum parallel strip-mine lengths at -O3 395

Table 52 Maximum vector strip-mine lengths at -O3 400

Table 53 Four processor system strip lengths 400

Table 54 ASCII character set 403

Table 55 Unformatted Cray files readable by CONVEX FORTRAN 413

How to use this guide

Purpose and audience

This guide tells you how to use the CONVEX FORTRAN compiler. Subjects discussed include compiling, loading, and executing programs, as well as a thorough discussion of the syntax of the language, including ANSI FORTRAN and CONVEX extensions. Other pertinent information includes input/output operations, error processing, program optimization, utility libraries, debugging, and various compatibility modes.

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.

If you are unfamiliar with the ConvexOS operating system, also consult the bibliography. Although a detailed knowledge of the operating system is not necessary for an understanding of this document, some familiarity with the system is beneficial.

Organization

This manual contains two separate, autonomous books: the *CONVEX FORTRAN User's Guide* and the *CONVEX FORTRAN Language Reference Manual*, as well as sections 1F and 3F of the online manual pages. The book is organized as follows:

- The front matter of this guide contains a common table of contents, list of tables, and list of figures, all of which list the contents of both the User's Guide and the Language Reference Manual. It also contains this common preface.

- Part 1 contains the *CONVEX FORTRAN User's Guide* in its entirety. The information contained in the User's Guide is primarily concerned with compiling, loading and executing FORTRAN programs using the CONVEX FORTRAN compiler. This book 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 various optimizations available in the compiler.
  - Chapter 2 describes how to use the CONVEX input/output facilities. It describes I/O statements and their parameters, file specifications, record structures, and record access.
  - Chapter 3 discusses how to use character data: building character substrings, using character constants, declaring character data, initializing character variables, and using character library functions.
  - Chapter 4 presents an overview of the tools available for debugging and analyzing FORTRAN programs.
  - Chapter 5 describes the CONVEX FORTRAN calling conventions and describes how to call routines written in languages other than FORTRAN.
  - Chapter 6 describes the use of ConvexOS system utilities.
  - Chapter 7 discusses runtime error processing and describes how the runtime library processes errors, what the defaults are, and how to override the defaults.
  - Appendix A describes the data types supported by CONVEX FORTRAN and shows their internal representations.
  - Appendix B lists error and advisory messages that can occur during compilation or at runtime.
  - Appendix C lists and describes the runtime library and routines.
  - Appendix D describes CONVEX FORTRAN compatibility with the IEEE 754 standard.

- Part 2 contains the *CONVEX FORTRAN Language Reference Manual* in its entirety. The Language reference manual provides a working definition for the CONVEX FORTRAN programming language, and encompasses the American National Standard programming language FORTRAN, ANSI X3.9-1978. It includes the features of the American National Standard programming language Fortran 90, ANSI X-3.198-199x, which have been implemented in this version of CONVEX FORTRAN. This book is organized as follows:
  - Chapter 1 discusses FORTRAN program elements and program unit format.
  - Chapter 2 discusses constants, variables, arrays, expressions, and function references.
  - Chapter 3 describes specification statements.
  - Chapter 4 discusses the DATA statement.
  - Chapter 5 describes the assignment statement and defines values used in a program.
  - Chapter 6 describes functions and operations of control statements.
  - Chapter 7 discusses files, units, Input/Output (I/O) statement components, data transfer I/O statements, and auxiliary I/O statements.
  - Chapter 8 defines format specification descriptors and carriage-control options and separators.
  - Chapter 9 discusses functions and operations of subprograms.
  - Chapter 10 discusses the BLOCK DATA statement.
  - Appendix A lists generic and intrinsic functions in table form, including the number of arguments, generic name, specific name, type of argument, and type of result.
  - Appendix B describes the preprocessor.
  - Appendix C describes compiler directives.
  - Appendix D lists maximum sizes for various elements in a FORTRAN program.
  - Appendix E lists the ASCII character set.
  - Appendix F discusses FORTRAN 66 compatibility.
  - Appendix G discusses Cray FORTRAN compatibility.

- Appendix H discusses VAX FORTRAN compatibility.
- Appendix I discusses Sun FORTRAN compatibility.

- Part 3 of this Guide contains hard copy versions of the online man pages, sections 1F and 3F. This part is organized as follows:
  - Section 1F of the man pages includes those pages that describe the CONVEX FORTRAN compiler (`fc`) and utilities provided with it.
  - Section 3F of the man pages includes those pages that describe the library functions contained in the CONVEX FORTRAN utility library.

An index covering all three parts is included at the end of this Guide.

Scope

This manual covers CONVEX FORTRAN Version 7.0, which runs under ConvexOS Version 9.1 or higher. The CONVEX FORTRAN compiler runs on all CONVEX hardware platforms, including C1, C2 and C3 Series architectures.

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.
- *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, filenames, 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.

The following conventions apply to the Language reference manual only:

- *Text describing CONVEX extensions to the FORTRAN language appear in this type style.*
- Command sequences or forms which are CONVEX extensions appear in *`monospaced italics, like this text.`*

Notes

Notes are set apart from standard text. An example is shown below.

Note

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

Associated documents

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

- *ConvexOS Primer* (DSW-133) has basic self-instruction for learning and using the ConvexOS operating system.
- *ConvexOS Man Pages for Programmers* (DSW-332) contains copies of Sections 2 through 5 of the online man pages. These man pages are primarily concerned with operating system information for programmers.
- *ConvexOS Man Pages for Users* (DSW-331) contains copies of Sections 1 and 7 of the online man pages. These man pages are primarily concerned with operating system information for users.

- *CONVEX* `adb` *Debugger User's Guide* (DSW-009), a tutorial and reference manual, describes the functions and operations of the CONVEX `adb` debugger.
- *CONVEX Consultant User's Guide* (DSW-025) (optional product) describes the functions and operations of the CONVEX `csd` debugger, postmortem dump (`pmd`) utility, and the `gprof` profiler.
- *CONVEX Compiler Utilities User's Guide* (DSW-096) describes the CONVEX loader.
- *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 Performance Analyzer (CXpa) User's Guide* (DSW-251) (optional product) describes how to use the interactive profiler at the various levels that are available.
- *CONVEX CXdb Concepts* and the *CONVEX CXdb Reference* describe all aspects of the optional CXdb debugger, which is briefly described in Chapter 4 of the CONVEX FORTRAN User's Guide.
- *CONVEX FORTRAN Optimization Guide* (DSW-034) describes the different types of optimization available in CONVEX FORTRAN and shows you how to use optimization directives and options.

Other documents of interest include the following:

- American National Standard programming language FORTRAN, ANSI X3.9-1978. This book is the definition of standard FORTRAN 77, which is fully supported in CONVEX FORTRAN V7.0.
- American National Standard programming language Fortran 90, ANSI X-3.198-199x. This book is the definition of standard Fortran 90, certain features of which are supported in CONVEX FOTRAN V7.0.

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.

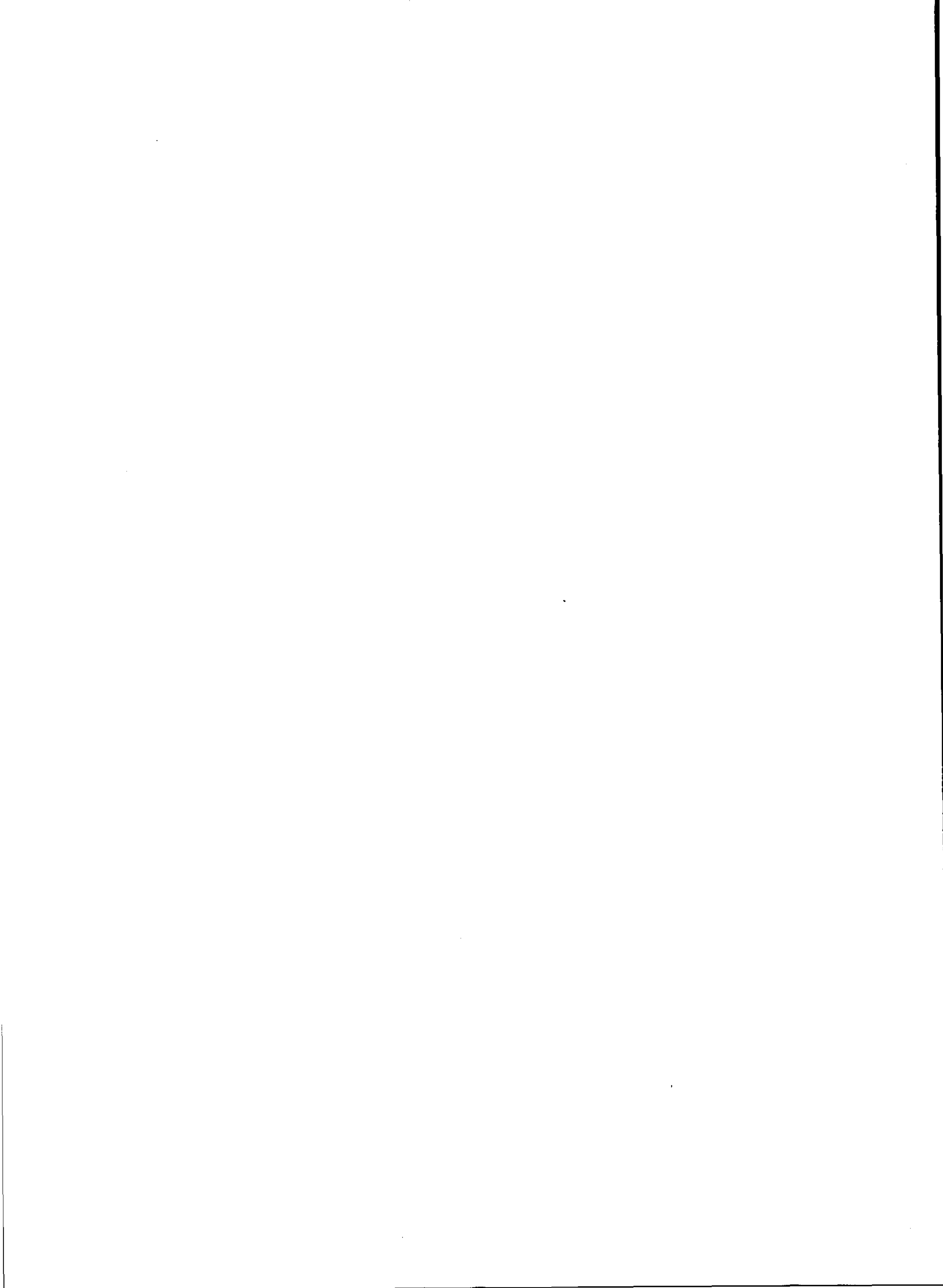
Using `contact` requires:

- A UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC.
- The full path name or the program or utility in question.
- The version number of the program or utility in question.

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

Part 1
CONVEX FORTRAN User's Guide

This part contains the *CONVEX FORTRAN User's Guide* in its entirety. The information in this part is primarily concerned with compiling, loading and executing FORTRAN programs using the CONVEX FORTRAN compiler.



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

Overview

The CONVEX FORTRAN compiler translates a source file containing one or more FORTRAN program units into an object module. The object module can then be linked with library routines or other object modules for execution on a CONVEX computer. Previously compiled programs written in CONVEX assembly language, C, or Ada 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 family of supercomputers. In addition, use of optimization options causes the compiler to optimize, vectorize, and parallelize source code to efficiently maximize execution efficiency.

The CONVEX FORTRAN compiler has options for requesting Cray FORTRAN, VAX FORTRAN, or Sun FORTRAN compatibility, IEEE-standard representation of floating-point values, and inline substitution of subroutines. IEEE functionality requires that the target machine be equipped with IEEE support hardware.

To enhance compilation speed, the CONVEX FORTRAN compiler generates object code directly. Another option, however, produces assembly-language code, which can be inspected, modified, or assembled directly.

To assist in program checkout, the compiler interfaces with several debuggers and utility routines, including a postmortem dump utility, a source-level debugger, and an assembly-level debugger.

CONVEX FORTRAN programs can also be compiled under COVUEshell. COVUEshell is a CONVEX product that provides a VMS-like interface and supports many of the Digital Command Language (DCL) commands. For further information, refer to the *CONVEX COVUEshell Reference Manual.*

File-naming conventions

The compiler distinguishes a file type by the extension added to the end of the file name. A FORTRAN source file is identified either by the extension `.f` or by the extension `.FOR`. The compiled object file has the same name as the source file except that it ends with `.o`.

When you compile and load a single source file during one invocation of the compiler, the object file is normally deleted. Unless you specify otherwise on the compiler command line, the executable module produced by the loader is placed into the file `a.out`.

Although the compiler normally produces object code directly, you can request that symbolic assembly code be generated. In this case, the name of the file that is produced is the same as that of the source file except that it ends with `.s`.

The following table summarizes the file naming conventions used by CONVEX FORTRAN.

File names for...	End with the extension...
FORTRAN source files	`.f or .FOR`
Object files	`.o`
Symbolic assembly-language files	`.s`
Inline intermediate files	`.fil`

Compiling programs

The `fc` command line has the following form:

```
fc [options] files [loader-options]
```

On the command line, *options* is one or more of the compiler options described in the following sections. Any options contained in an `OPTIONS` statement within a program override those specified on the command line.

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

The parameter *loader-options* is one or more loader options as described in the *CONVEX Loader User's Guide.* If specified, these options are passed to the loader when compilation is complete.

Language-compatibility options

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

`-cfc`

Causes the compiler to accept and use certain syntax, defaults, and features of the Cray FORTRAN language. Before using `-cfc`, read Appendix G of the *CONVEX FORTRAN Language Reference Manual.* (This option cannot be used with the `-i` or `-r` options.)

`-dfc`

Use `-dfc` only with the `-vfc` option and only when you use the Binary Data File Format Conversion feature discussed in Chapter 7 of the *CONVEX FORTRAN Language Reference Manual.* This option causes all references to VAX records in an I/O statement to be decomposed.

`-F66`

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

`-f90`

Allows use of the Fortran 90 features included in CONVEX FORTRAN Version 7.0. These include some Fortran 90 intrinsic functions (refer to Appendix A of the *CONVEX FORTRAN Language Reference Manual*) and Fortran 90 array notation (refer to Chapters 2 and 5 of the *CONVEX FORTRAN Language Reference Manual*).

`-sa`

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 applications 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 I of the *CONVEX FORTRAN Language Reference Manual.*

`-vfc`

Causes the compiler to accept and use certain syntax, defaults, and features of the VAX FORTRAN language. Before using this option, read Appendix H of the *CONVEX FORTRAN Language Reference Manual.*

Optimization options

The options listed in this section enable various optimizations.

`-ds`

Causes the compiler to automatically select loops to replicate and to compile several versions of such loops. The compiler then dynamically selects the version of each loop to be executed. This option is available only at optimization level `-O2` or `-O3`.

`-ep` *n*

Specifies the expected number of processors (*n*) on which the program is going to run. The value of *n* should be an integer from 1 to 4.

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.

-il

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

-is *directory*

Instructs the compiler to 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.

-no

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

-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 *CONVEX FORTRAN Optimization Guide* for more information.

-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 *CONVEX FORTRAN Optimization Guide* for more information.

-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	O0 optimizations plus program unit machine-independent scalar optimization
-O2	O1 plus vectorization
-O3	O2 plus parallelization

No machine-independent optimization is performed if this option is not specified.

`-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. `-peel` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can greatly lengthen compiler time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-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. `-ptst` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-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. `-ptstall` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-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`.

`-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 loops. This option is available only at optimization level `-O2` or `-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`.

`-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.

Note

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

`-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.

`-in`

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. This option can appear in the `OPTIONS` statement.

`-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.

`-p8`

Specifies that `INTEGER`, `LOGICAL`, and `REAL` variable declarations 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` variable declarations 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.

`-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 between real and complex and double-precision data is not maintained as specified by the ANSI-77 standard.

note

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

`-re`

Causes the compiler to generate reentrant code for parallel or recursive invocation of subprograms. This option makes it possible to call subroutines from parallel loops.

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

If you compile a program using the `-re` option, you must initialize all local variables that are to remain unshared.

`-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.

`-tm` *target*

Specifies the target machine architecture for which compilation is to be performed. *target* can take the value `c1`, `c2`, `c32`, `c34` or `c38`.Use `c1` when compiling for a C1 series machine; use `c2` when compiling for a C2 series machine; use `c32` when compiling for a C3200 series machine; use `c34` when compiling for a C3400 series machine; use `c38` when compiling for a C3800 series machine. If you specify a target machine, its instruction set is used regardless of the machine on which the compiler is running. If you do not specify a target machine, the compiler generates instructions for the class of machine on which it is running.

Note

The `file` utility may indicate that an executable generated with `-tm` specifying some C2 or C3 series machine is a c1 executable. This is because `file` only checks for instruction set differences between the two machines. An executable generated with the `-tm` option specifying a C2 or C3 series machine 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.

`-a1`

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. The `-a1` option can be used in the `OPTIONS` statement.

`-cs`

Compiles code to check that each subscript is within its array bounds. Does not check the bounds for arrays that are dummy arguments for which the last dimension bound is specified as * or 1. The `-cs` option can be used in the `OPTIONS` statement.

`-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 flag can be used with all levels of optimization. Because CXdb does not support debugging of inlined code, `-cxdb` cannot be used with the `-il` or `-is` options. Also, because CXdb requires object modules to be generated directly by the compiler, `-cxdb` cannot be used with the `-S` or `-pb` options.

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 4, "Program development tools," or to the *CONVEX CXdb User's Guide* for more information. CXdb is an optional product.

`-db`

Produces additional information for use by the symbolic debugger, `csd`, and passes the `-lg` option to the loader. This option can be used with all levels of optimization. If the `-O`*n* option is specified, there may be source statements for which no debugging information is generated for `csd`.

`-dc`

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

`-p`

Produces monitor routines for the `prof` profiler. The `prof` profiler is part of the optional CONVEX Consultant package.

`-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 part of the optional CONVEX Consultant package.

`-pa`

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 flag, `-pab`.

`-pab`

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 `-pa` flag. The performance analyzer (CXpa) is an optional CONVEX product.

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

`-par`

Generates routine-level instrumentation for profiling under CXpa.

`-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.

`-LST`

Generates a listing of compiled source program, inserts compiler error flags (if any) at the appropriate points, and writes it to `stdout`. Printer must handle 90 columns or more, 60 lines or more, and understand ASCII formfeeds.

`-LSTI`

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

`-na`

Suppresses all advisory diagnostic messages.

`-nv`

This option is no longer supported. Use `-or none` instead.

`-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`, or `array`. If this option is not specified, only the loop table is displayed.

`-xr`

Invokes the Cross-referencer and generates a cross-reference report at compilation time.

`-xra`

Invokes Cross-referencer and generates data file but does not generate report. The report can be generated by running `fcxref`. Refer to Chapter 4, "Program development tools," for more information on `fcxref`.

The following options can be used with `-xr` and `-xra`:

Option	Description
`-iw` *n*	Specifies the desired identifier width. Default is *n* = 16.
`-pl` *n*	Specifies a default maximum page length of *n* lines per page. Default is *n* = 60.
`-pw` *n*	Specifies a desired page width of *n* columns per line. Data that exceeds this width is wrapped to the next line. Default is *n* = 80.
`-xrf` *file*	Use *file* as the cross-referencer data file instead of `.fcxrefData`.
`-xrr` *file*	Sends the cross-reference report to *file* rather than to `stdout`.

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 $n = 8$.

`-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`.

`-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.

Note

These options, with the exception of `-fpp`, are being phased out. Refer to Appendix B of the *CONVEX FORTRAN Language Reference Manual* for more information.

`-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 B of the *CONVEX FORTRAN Language Reference Manual* 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`*dir*
: For `#include` files whose names do not begin with "/", the preprocessor searches first in the directory of the file argument, then in directories named in `-I`, then in directories on a standard list. No more than eight directories can be specified.

`-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 B of the *CONVEX FORTRAN Language Reference Manual* for details on the use of this option.

Miscellaneous options

The options listed in this section are not otherwise categorizable.

`-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`*string*

Finds the substitute compiler (`fskel`, `fpp`, and `errmsgf`) in the directory named *string*. The default directory is `/usr/convex/oldfc`, which contains the previous default compiler for use as a backup.

`-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.

Note

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

`-nosc`

Disables short circuit evaluation of conditionals. See Chapter 6 of the *CONVEX FORTRAN Language Reference Manual* for more information.

`-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.

`-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.

`-tl` *n*

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

`-vn`

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

`-72`

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 flag 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 must process the resulting object code (`ld`) before it can be executed. 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 (`fc`) command 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.

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 result during compilation and at runtime. The messages are grouped into the following categories:

- Compiler messages
- Optimization report
- Runtime messages

Compiler messages

The compiler may produce various error, warning, and informational messages. These messages are directed to the standard error file (`stderr`). 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 line of 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. 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).

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, an array table, or both. You can specify which tables are to be included in the optimization report by means of the `-or` compiler option.

Loop table, part 1

The loop table lists the optimizations that were performed on each loop and, if appropriate, the reasons why a possible optimization was not performed. The loop table can consist of one or two parts. Part 1 is always printed and contains the following information:

- `Line Num.`: Specifies the source line of the beginning of the loop. If the line number has two parts separated by a hyphen, the second part is the distribution number (due to loop distribution).
- `Iter. Var.`: Specifies that the name of the iteration variable controlling the loop or `*NONE*`. If the iteration variable has two parts separated by a colon, the second part is the inline substitution instance of that variable.
- `Reordering Transformation`: Indicates which reordering transformations were performed. A reordering transformation does not eliminate operations from a program or replace them with simpler operations, but rearranges them so they can be more efficiently executed. This column has one of the following values:

Value	Explanation
`Scalar`	No transformation of this type was performed.
nn`% VECTOR`	The loop was partially vectorized, with the percentage (*nn*) specified being executed in vector mode.
`FULL VECTOR`	The loop was fully vectorized, with all operations being executed in vector mode.
`PARALLEL`	The loop runs in parallel mode.
`PARA/VECTOR`	The loop was vectorized, and the strip mine loop runs in parallel mode.
`Dist`	Loop distribution was performed.
`Inter`	Loop interchange was performed.

- `Optimizing/Special Transformation`: Indicates which optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with simpler operations. A special transformation allows the compiler to vectorize or parallelize code under special circumstances. This column has one of the following values:

Value	Explanation
`Unroll`	The loop was completely or partially unrolled.
`Reduction`	The compiler recognized a reduction and vectorized the loop.

Value	Explanation
`Pattern`	The compiler recognized a special pattern and vectorized the loop.
`Synch`	The compiler inserted synchronization code to ensure correct execution of a parallel loop.

- `Mode`: If used, this column refers to multiple execution modes controlled by dynamic selection. This column can contain the following values:

Value	Explanation
`S`	Specifies scalar execution.
`V`	Specifies vector execution.
`P`	Specifies parallel execution.
`Z`	Specifies parallel-outer execution, vector-inner execution.

Loop table, part 2

Part 2 of the loop table is only printed if there is relevant information to be shown. Part 2 of the loop table includes the following information:

- `Line Num.`: Specifies the source line of the beginning of the loop.
- `Iter. Var.`: Specifies the name of the iteration variable controlling the loop or `*NONE*`.
- `Analysis`: Indicates why a transformation or optimization was not performed, or additional information on what was done.

Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed. The array table has the following information:

- `Line Num.`: Specifies the source line on which the reference occurs.
- `Var. Name`: Specifies the name of the array being referenced.
- `Optimization`: Contains one of the following values:

Value	Explanation
`Hoist`	The vector load was found to be loop invariant and was moved outside the loop.
`Sink`	The vector store was found to be loop invariant and was moved outside the loop.

- `Dependencies`: Shows the names of other variables in a recurrence, in the form *name*@*linenumber*. If the reference could be to any memory location, it is in the form `*MEM*`@*linenumber*.

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. Refer to Appendix B, "Compiler and runtime messages" for a list of error numbers and explanations.

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.

Program interfaces

Calling sequences for the runtime system include the use of the short-form 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.

Note

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

The runtime system provides scalar and vector versions of the math intrinsics. The compiler determines which version of the routine to call.

If you code part of your program in a language other than FORTRAN, such as C or assembly language, you must use the same names that are output by FORTRAN or the program cannot link properly. CONVEX FORTRAN uses the following naming conventions:

Main program:	`_MAIN__`	(1 preceding underscore and 2 following)
Blank common:	`___blnk_`	(3 preceding underscores and 1 following)
Named common:	__*name*_	(2 preceding underscores and 1 following)
Subprogram:	_*name*_	(1 preceding underscore and 1 following)

where *blnk* and *name* represent the symbolic name.

To get names generated by FORTRAN, you must append and prefix the appropriate number of underscores. For example, because C prefixes a single underscore to function names and external variable names, you must append a single underscore to function names and external variable names. Thus, a call to the FORTRAN routine `FFT` from C must be written `fft_`.

If your main program is not in FORTRAN, FORTRAN units 5, 6, and 0 are not connected and data format conversions will not work. Signal handling depends on the language used in the main program.

Optimization

The CONVEX FORTRAN compiler offers several types of optimization that produce efficient code and enhance the execution speed of a program. You can specify the optimization to be performed on your program with command line options, compiler directives, and the `OPTIONS` statement.

The types of optimization that can be performed, along with their respective command line options, are shown in Table 1.

Table 1
Optimization levels

Option	Description
-O0	Basic block machine-independent scalar optimization.
-O1	Basic block and program unit machine-independent scalar optimization.
-O2	Vector optimization.
-O3	Parallel optimization.

If you do not specify an optimization level, the compiler defaults to the -no (machine-dependent scalar) optimizations. Each optimization level is cumulative; it retains the optimizations of the previous level.

This section presents an overview of the above automatic optimizations available in the CONVEX FORTRAN compiler, as well as semi-automatic optimization techniques. For detailed descriptions and examples, refer to the *CONVEX FORTRAN Language Reference Manual* and the *CONVEX FORTRAN Optimization Guide.*

Machine-dependent optimization

Machine-dependent optimization enhances the object code produced by the compiler to take advantage of the machine architecture. Types of machine-dependent optimization include instruction scheduling, register allocation, and branch optimization.

The compiler always performs machine-dependent optimization regardless of the optimization level.

Local scalar optimization

Local scalar optimization is machine-independent optimization performed on a sequence of code with one entrance and one exit. Local optimization uses information within the source code to eliminate unnecessary computations during program execution. The types of local optimization that are performed include assignment substitution, common subexpression elimination, and algebraic simplification.

To request local scalar optimization, compile your program with the -O0 command line option.

Global scalar optimization

Global scalar optimization is machine-independent optimization performed over an entire program unit—in particular, conditional statements and loops. Types of global scalar optimization that are performed include dead code elimination, code motion, copy propagation, partial redundant subexpression elimination, and strength reduction.

To request global scalar optimization, compile your program with the -O1 command line option. When you request global scalar optimization, the compiler also performs local scalar optimization.

Vectorization

Vectorization causes the processor to use vector registers that can manipulate up to 128 elements of an array with a single instruction. To vectorize a loop within a program, the compiler converts scalar operations on data arrays into equivalent vector operations. Some of the techniques that the compiler uses to perform vectorization include strip mining, loop distribution, and loop interchange.

To request vectorization, compile your program with the -O2 command line option. When you request vectorization, the compiler also performs global scalar optimization and local scalar optimization.

Parallelization

Automatic parallelization attempts to generate parallelized code for all eligible loops in the program. This code allows the processor to execute independent, nonsynchronized iterations of a loop in parallel on separate CPUs.

A loop can be parallelized if there are no dependencies between iterations, that is, if the results computed by one iteration do not depend on the results of earlier iterations. A loop cannot be parallelized if it includes:

- Loop-carried dependencies
- Exits
- Calls to subroutines

To request parallelization, compile your program with the `-O3` command line option. When you request parallelization, the compiler also performs vectorization, global scalar optimization, and local scalar optimization.

Optimization level `-O3` can reduce the amount of elapsed time taken by a program, but it usually increases the amount of CPU time because of the additional work needed to coordinate the processors. Optimization level `-O2` is often more appropriate for moderate-sized programs that run under heavy timesharing.

Inline substitution

Inline substitution (inlining) replaces a subroutine or function call with the actual body of the subprogram. The inlined code can then be optimized. During the substitution, actual arguments are mapped to dummy arguments and local identifiers are assigned unique names.

Before performing inline substitution, you must first compile each subroutine to be inlined with the `-il` command line option. The `-il` option creates an intermediate-language (`.fil`) file for the subroutine.

To perform the inline substitution, compile your program with the `-is` command line option. The `-is` *directory* option causes the compiler to perform inlining on every subroutine in the specified *directory* for which a `.fil` file exists.

Loop replication

Loop replication causes the compiler to perform either loop unrolling or dynamic loop selection on all eligible loops in the program. You can individually select either loop unrolling or dynamic loop selection.

Loop unrolling reduces loop overhead by duplicating the body of the loop and can be performed on scalar and vector loops.

Dynamic loop selection creates multiple versions of a loop and to generate code that selects, at runtime, which version to execute. As appropriate, the compiler generates scalar, vector, parallel, and parallel-vector versions of a loop. Parallel-vector means that the loop is strip-mined with the strip-mine loop running in parallel.

To request loop replication, compile your program using the -O2 (or -O3) command line options with -rl, -ur, or -ds. You can select individual loops for replication with the UNROLL and SELECT directives.

IF-DO optimizations

IF-DO optimizations are performed at -O2 and -O3 optimization levels. In general, they modify loops containing tests to improve vector performance. Tests can be promoted out of their containing IF or DO loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of masked vector instructions that must be executed, thereby improving performance. Refer to the *CONVEX FORTRAN Optimization Guide* for a complete discussion of each IF-DO optimization.

IF-DO optimizations consist of the following:

- Redundant index test elimination, in which the compiler recognizes tests against some index variable that always evaluate to .TRUE. or .FALSE. and eliminates the code.
- Loop peeling, in which a test that always evaluates to .TRUE. or .FALSE. is contained in the first and/or last iteration of the loop and may therefore be relocated to outside the loop. Refer to the -nopeel, -peel, and -peelall options discussed earlier in this chapter.
- Test promotion, in which a test can be promoted out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals (or no tests at all), so they execute much faster. Multiple tests can be promoted, with loop replications made for each. In nested loops, tests are promoted to the level of the outermost nest, and all subordinate loops are replicated. Refer to the -noptst, -ptst, and -ptstall options discussed earlier in this chapter.

Input/output operations

2

This chapter describes the particulars of FORTRAN input/output operations as they are performed under CONVEX FORTRAN and presents specific information about ConvexOS files.

Units

The unit number in a CONVEX FORTRAN input/output statement can range from 0 through 255. A unit number can be specified either explicitly or implicitly. The following WRITE statement, for example, specifies an explicit unit of 9.

```
WRITE (9, 100) I, X, Y
```

In certain forms of the READ and WRITE statements and in statements such as ACCEPT, PRINT, or TYPE, the unit number is implicit. Table 2 shows the general forms of CONVEX FORTRAN I/O statements in which the unit is specified implicitly. In the table, *f* indicates the FORMAT statement number and *list* indicates the data to be transferred.

Table 2
Implicit units

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

By default, a program looks at unit 5 for input, writes output to unit 6 (standard output), and sends error messages to `stderr` (standard error). 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.

By default, CONVEX FORTRAN assigns unit 5 to *`stdin`*, unit 6 to `stdout`, and unit 0 to `stderr`. If 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.

You can reopen units 5, 6 and 0, but the C language pointers *`stdin`*, `stdout` and `stderr` are not reassigned.

Logical names

Every unit in CONVEX FORTRAN, except unit 0, is associated, by default, with a logical name of the form `FOR`*nnn*, where *nnn* is the unit number. This logical name is used to create a default ConvexOS file name in the form *`fort.nnn`* to which the unit is automatically preconnected. Preconnected means that the file is connected to the unit when the program begins executing and can be referenced by input/output statements without prior execution of an `OPEN` statement.

The following statement opens and writes to the file `fort.35`.

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

Table 3 shows examples of units, the logical names associated with the units by default, and the ConvexOs file names to which they are preconnected by default.

Table 3
Examples of default logical names

Unit	Default logical name	Default file name
8	FOR008	fort.8
52	FOR052	fort.52

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

Example:

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

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

The following statement generates a file named `fort.4`.

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

The files `stderr`, `stdin`, and `stdout` can also be redirected from the command line as described in the description of the C shell (`csh`) in the *ConvexOS Man Pages for Users*.

The OPEN statement

The OPEN statement connects a unit to a file so that any subsequent input/output operations to that unit access the specified file. An OPEN statement can contain a FILE= clause whose value can either be a logical name or file name. The value specified in the FILE= clause overrides the default logical name or file name associated with the unit.

Note

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

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

Examples:

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

```
OPEN (94)
```

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

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

The following example opens the specified file.

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

Assigning logical names

You can customize your ConvexOS environment to assign FORTRAN logical names to files. You can also change the default name assigned to scratch files by changing the environment variable FORTEMP. For a discussion of the ConvexOS environment, refer to the *ConvexOS Primer* or to the *ConvexOS Man Pages for Users*.

In the most commonly used command interpreter, the C shell (csh), the commands setenv and unsetenv control the setting and resetting of variables in your working environment. The format of these commands is

setenv *name value*

unsetenv *name*

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

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

Examples

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

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

```
setenv FORTEMP MYFILE
```

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

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

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

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

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

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

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

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

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

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

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

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

Forms of input/output

CONVEX FORTRAN supports formatted, list-directed, namelist-directed, and unformatted input/output (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.

Although similar to formatted statements in function, list-directed and namelist-directed I/O statements 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. But, a formatted, list-directed, or namelist-directed I/O statement can transfer more than one record.

File type

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.

An internal file that is a single character variable, array element, or substring consists of one record whose length is the same as that of the character variable, array element, or substring. An internal file that is a character array has a sequence of records, each of which is a single array element. The order of subscript progression determines the record sequence in an internal file.

Before data is transferred, an internal file is always positioned at the beginning of the first record.

Access modes

The method for retrieving and storing records in a file is the access mode, which is specified by each I/O statement. CONVEX FORTRAN supports sequential and direct access modes.

Sequential

Sequentially accessed records are written to or read from the file starting at the beginning and continuing through the file, one record after another. You can access a particular record only after all the records preceding it have been read. You can write new records only at the end of the file.

Example:

```
READ (10,*) A, B
```

In this example, the READ statement causes the next two real values to be read into variables A and B.

Direct

This mode allows you to choose the order in which records are read and written. Each READ or WRITE statement must include the record number.

Examples:

```
WRITE (10,REC=28) I
WRITE (10,REC=15) J
```

The first statement writes the value of I to record 28, and the second statement writes the value of J to record 15.

Logical records

The definition of a logical record depends on the combination of the I/O form and the mode specified by the FORTRAN I/O statement. Each execution of a FORTRAN unformatted I/O statement causes a single logical record to be read or written. Each execution of a FORTRAN formatted I/O statement causes one or more logical records to be read or written.

Direct-access external file

A logical record in a direct-access external file is a string of bytes, the length of which you specify when you open the file. `READ` and `WRITE` statements must not try to access more data than fits into one record. Shorter logical records are allowed. Unformatted direct `WRITE` statements leave the unfilled part of the record undefined. Formatted direct `WRITE` statements pad the unfilled record with blanks.

Sequential-access external file

A logical record in a sequentially accessed external file can be of any length. The size of the items in the list of I/O values (the I/O list) determines the logical record length for unformatted sequential files. For formatted `WRITE` statements, the format statement interacting with the I/O list at execution time determines the logical record length. Formatted sequential access causes one or more logical records ending with the "newline" (hexadecimal 0A) character to be read or written.

Namelist-directed input/output

Namelist-directed I/O statements are similar in function to list-directed statements. For variable-length files, the namelist-directed statement uses data types instead of explicit format specifiers to control data translation and formatting. For fixed-record-length files, the namelist-directed statement reads and writes records of a fixed length.

Namelist-directed I/O cannot read files written with nonblank carriage-control characters in column 1.

Internal files

The logical record length for an internal `READ` or `WRITE` is the length of the character variable or array element. Thus, a simple character variable is a single logical record.

Data file formats

In addition to fixed-length record formats, CONVEX FORTRAN supports formatted and unformatted variable-length record formats.

- Formatted, variable-length records end with a linefeed character.
- Unformatted, variable-length records begin and end with a field of 4 bytes that contains the record length. Files written using Cray data format (with `FORM=UNFORMATTED/CRAY` in the `OPEN` statement) begin with a field of 8 bytes that contains the record length in 8-byte words. Files written with the Cray format have no ending field. A sign bit can be set.

CONVEX FORTRAN supports Cray sequential access unformatted unblocked data files, direct access unformatted unblocked data files, and, through use of the `fcUnblock` utility, sequential access unformatted blocked data files. Using the Cray data format does not necessarily mean compiling with the `-cfc` flag; refer to Chapter 7, "Input/output statements," of the *CONVEX FORTRAN Language Reference Manual* for more information on data formats.

Accessing file pointers

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

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

Example:

FORTRAN program:

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

C routine:

```
#include <stdio.h>
#include <fcntl.h>
void bypasscache_(fp_ptr) /* note trailing
underscore
                             required because
routine
                             is called by a
FORTRAN
                             program
*/

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

Input/output statement summary

Table 4 summarizes the input/output statements available in CONVEX FORTRAN. The *CONVEX FORTRAN Language Reference Manual* describes these statements in detail.

Table 4
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.
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`.
	`ENCODE`	Transfers data between arrays or variables in internal storage and translates the data from internal to character form.
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.

The `ACCEPT`, `DECODE`, `TYPE`, `ENCODE`, and `FIND` statements are CONVEX extensions to the FORTRAN 77 standard.

Character data

3

This chapter describes how to use character data, how to build character strings, substrings, and constants, how to declare character data, initialize character variables, manipulate data for longer arguments, and build character library functions.

Character constants

A character constant is a string of characters enclosed in apostrophes or quotation marks. A space is a valid character. To include an apostrophe as part of a character constant, use two consecutive apostrophes with no intervening blanks. The following examples show how to assign a character value to a character variable.

Examples:

```
STRING = ' ab c'    ! or STRING = " ab c"
ABC = 'BAR'         ! or ABC = "BAR"
CANNOT = 'CAN''T'   ! or CANNOT = "CAN'T"
QUOTE = '"'         ! or QUOTE = """"
```

If the size of the variable is smaller than the number of characters being assigned to the variable, the string is truncated on the right; if the size of the variable is larger than the number of characters being assigned, the string is padded on the right with blanks up to the designated length of the variable.

Example:

```
CHARACTER*2 ABC
ABC = 'BAR'      ! The value BA is stored in ABC

CHARACTER*6 ABC
ABC = 'BAR'      ! The value BAR^^^ (^ = blank) is
                 ! stored in ABC
```

You can use the PARAMETER statement to give character constants symbolic names.

Example:

```
CHARACTER*(*) POEM
PARAMETER (POEM = 'BEOWULF')
```

You can now use the symbolic name POEM anywhere a character constant is allowed.

The variable on the left side of a character assignment statement can not appear on the right side. The same name can appear on both sides of the assignment statement if it is part of a substring reference and the substring ranges are distinct.

Example:

```
CHARACTER*30 A
A(1:10) = A(5:15)    ! Invalid
A(1:10) = A(20:30)   ! Valid
```

Declaring character variables

The CHARACTER statement declares character variables or arrays as shown in the following examples:

```
      CHARACTER*8 GAME(10), TENNIS
C     Declare the array GAME with ten 8-character
C     elements and the variable TENNIS, which is 8
C     characters long

      CHARACTER*8 TEAM, POLO*2, GAME
C     Declare TEAM and GAME as 8-character
C     variables  and POLO as a 2-character variable
```

For more information on the CHARACTER statement, refer to the *CONVEX FORTRAN Language Reference Manual*.

Initializing character variables

The DATA statement initializes a character variable as shown in the following examples.

```
CHARACTER*8  GAME(10), TENNIS
DATA GAME(1), TENNIS /'HOCKEY', 'CONNORS'/

CHARACTER*10 TEAM
DATA TEAM /'COWBOYS'/
```

You can also initialize character variables in the CHARACTER declaration statement, as shown in the following example:

```
CHARACTER*10 TEAM /'COWBOYS'/
```

If necessary, the value used to initialize the variable is extended with blanks or truncated in the same manner as for the assignment statement.

Character substrings

A character substring is a portion of a character string and contains the name of a character variable or array element followed by delimiters that define the leftmost and rightmost characters in the substring. Substrings have the following forms:

```
      v ([e1]:[e2])
C     Substring of a character variable
```

or

```
      a (s,s ...) ([e1]:[e2])
C     Substring of an array element
```

where

v
: is a character variable name.

a (*s,s...*)
: is a character array element name.

e1, *e2*
: are integer expressions and are called substring expressions.

The value *e1* specifies the leftmost character position of the substring, and the value *e2* the rightmost character position. The leftmost character position is position 1. If *e1* is not specified, the

default value is 1; if *e2* is not specified, the default value is `LEN(a)`. Use these values to select certain segments (substrings) from a character variable or array element. For example, for the following character string:

```
POE = 'ONCE UPON A MIDNIGHT DREARY'
```

Various substrings can be extracted as shown in the following examples.

Examples:

```
POE(13:20)  ! Extracts the substring MIDNIGHT
POE(:11)    ! Extracts the substring ONCE UPON A
POE(13:)    ! Extracts the substring
            ! MIDNIGHT DREARY
```

Concatenating character strings

Use double slashes (//) to concatenate strings from two or more separate strings. Thus, to create a variable called `BUDGET` from the strings

```
'FOUR '
'BILLION '
'DOLLARS'
```

define each as a character variable with a specified length, as follows:

```
CHARACTER*20 BUDGET
CHARACTER*5  S1/'FOUR '/
CHARACTER*8  S2/'BILLION '/
CHARACTER*7  S3/'DOLLARS'/
```

Next, use the double slashes to create your new string:

```
BUDGET = S1//S2//S3   ! BUDGET = "FOUR BILLION
                      ! DOLLARS"
```

This string has all the values assigned to each of the substrings. You can select a given substring using a character substring reference. These character substring references can be used to access those portions of BUDGET containing the concatenated values:

```
BUDGET(:5)
BUDGET(6:13)
BUDGET(14:20)
```

Character input/output

The CHARACTER data type enables you to read and write character strings of any length.

Example:

```
      CHARACTER*20 HEADER
            .
            .
            .
      READ (6,100) HEADER
100   FORMAT (A)
```

The preceding code causes 20 characters to be read from unit 6 and stored in the variable HEADER.

Character library functions

There are eight character intrinsic functions and two character utility functions.

Character intrinsic functions	Utility functions
ICHAR	RINDEX
CHAR	LNBLNK
LEN	
INDEX	
LGE, LGT, LLE, LLT	

ICHAR

The ICHAR function returns the decimal value of a specified ASCII character. This function has the form

```
ICHAR (c)
```

where *c* is an ASCII character expression. If *c* is longer than one character, only the value of the first character is returned; the rest of the expression is ignored.

Example:

```
CHARACTER*5 STATE/'TEXAS'/
I=ICHAR(STATE(1:1))          ! sets I to 84, the
                             ! ASCII value of 'T'
```

CHAR

The CHAR function returns the ASCII character corresponding to a specified decimal value. CHAR returns an ASCII value from 0 through 255. This function has the form:

```
CHAR (i)
```

where *i* is an integer expression equivalent to an ASCII character code. Unlike FORTRAN, C strings are usually null terminated. You can get this effect in FORTRAN by concatenating a CHAR(0) at the end of the string.

Example:

```
CHARACTER*80 CSTR
PRINT *, CHAR(84)        ! prints the letter T
CSTR = 'ABC' //CHAR(0)  ! places a null at the
                         ! end of a string for C
```

LEN and LNBLNK

The LEN function returns an integer length to show the length of a character expression. This function is useful for finding the true length of an object declared CHARACTER *(*). The LEN function has the form:

```
LEN (c)
```

where *c* is a character expression.

Example:

```
I=len('A'//'B'//'C')    !sets I to 3
```

The LNBLNK function returns an integer value indicating the position of the right-most nonblank character:

```
I=LNBLNK('ABC^^^^')     !sets I to 3
```

INDEX and RINDEX

The INDEX function searches a specified string for the occurrence of a substring; the RINDEX function finds the last occurrence of a string.

If the substring exists, INDEX returns an integer value corresponding to the character position at which the substring begins. If no substring exists, INDEX returns 0. If the substring appears more than once, INDEX returns the starting position of the leftmost substring. This function can be used to search for specific characters, words, or sentences located in a given text. The INDEX function has the form:

```
INDEX (c1,c2)
```

where

c1
: is a character expression that specifies the string to be searched.

c2
: is a character expression representing the substring for which a match is desired.

Example:

```
CHARACTER*20 STRING /'NOW IS THE TIME'/
I=INDEX(STRING,'THE')              ! sets I to 8
J=INDEX(STRING,'TIME')             ! sets J to 12
R=INDEX(STRING,'I')                ! sets R to 5

INTEGER RINDEX,R
R=RINDEX(STRING,'I')               ! sets R to 13
```

Lexical comparison functions

Four intrinsic functions (LGE, LGT, LLE, and LLT) are used for comparing the standard lexical relationships of two character strings and returning a logical value of .TRUE. or .FALSE.

You can get the same results by using the arithmetic relational operators (such as `.GE.`) instead of the lexical functions because CONVEX machines store strings in ASCII.

Table 5 describes the lexical comparison functions.

Table 5
Lexical intrinsic functions

Function	Value is true if...
LGE(*c1*,*c2*)	The string *c1* follows or equals the string *c2* in the ASCII collating sequence.
LGT(*c1*,*c2*)	The string *c1* follows the string *c2* in the ASCII collating sequence.
LLE(*c1*,*c2*)	The string *c1* precedes or equals the string *c2* in the ASCII collating sequence.
LLT(*c1*,*c2*)	The string *c1* precedes the string *c2* in the ASCII collating sequence.

Program development tools

4

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`)
- Performance analyzer (CXpa)
- CONVEX Consultant
- CONVEX Visual Debugger (CXdb)
- `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
- Functions
- Subroutines
- Intrinsics
- COMMON block names
- Include files

- Cray pointers
- VAX structures and unions

Cross-reference information is generated through the use of one of two `fc` command line options, either `-xr` or `-xra`. Both options generate a data file containing all the cross-referencing information necessary to build the report. The default name for this file is `.fcxrefData`; this name can be changed through use of the `-xrf` command line option described in Table 6.

The options differ in that `-xr` 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. Thus, `-xr` is used to independently cross-reference individual source files. To globally cross-reference several FORTRAN source files, you can use the `-xra` option, which 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 6. `fcxref` options can be specified in the `FCOPTIONS` environment variable, where they are automatically used on invocation of the compiler or cross-referencer, whichever is appropriate.

Table 6
Cross-referencer options

Option	Command line	Description
`-xr`	`fc`	Invokes cross-referencer and generates report at compilation time.
`-xra`	`fc`	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`
	`fc -xra`
	`fcxref`
		Use *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`	Deletes the data file after generating the report. By default, the data file is saved when `fcxref` is run standalone.
`-xrr` *file*	`fc -xr,` `fcxref`	Sends the cross-reference report to *file* rather than to standard output.
`-xrg` *n*	`fc -xr,` `fcxref`	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. Defualt is $n = 8$.
`-iw` *n*	`fc -xr,` `fcxref`	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*.
`-pw` *n*	`fc -xr,` `fcxref`	Specifies a desired page width of *n* columns per line. Data that exceeds this width is wrapped to the next line. Default is $n = 80$.

Note that the `-sl` source listing option is no longer supported. Use `-LST` 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).

Example:

```
fc prog1.f -xr -xrg 10 -xrr prog1.xr -pw 78 -iw 12
```

This line would compile the program prog1.f and generate a cross-reference report that include common block reports using the exhaustive members list and differing members report formats, both ordered by member names. The report would be stored in the file prog1.xr, and would be 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.

Example:

```
fc prog1.f prog2.f prog3.f -xr -xrr progs.xr
```

This line would compile prog1.f, prog2.f and prog3.f, generate a cross-reference report for each one (using default report option values), and place 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.

Example:

```
fc prog1.f -xra -xrf prog1.xrdata
```

Here the -xrf option was used to specify 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 prog1.xrdata -xrg 10 -iw 12 -pw 78 -xrr progn.xr
```

The above command line generates a cross-reference report based on the data contained in the file prog1.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.

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; where appropriate, cross-reference report segments are shown.

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, number of common reports, whether to delete the data file, and whether `fcxref` is being run standalone or by the `fc` driver. See Figure 1.

Figure 1
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
delete data file => yes
stand alone run  => no
```

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, kind (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, and 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.

The right column of the routine report contains the specific object type (variable, array, parameter, label, main program, subroutine, function, intrinsic, 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*]

Where *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 7.

Table 7
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 is 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 2. This example, along with the examples in Figures 3, 4, 5, 7 and 6 were compiled with the folowing 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 2
Cross-referencer routine report

```
CONVEX FORTRAN CROSS-REFERENCE                                      PAGE: 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#
                 .
                 .
                 .
```

Figure 2 shows the symbol 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. The report 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 the "Include file reference table" section of this chapter), and assigned a value on line 4 of this include file.

$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. 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 last entry in the report tells us that MAIN, the main program name, appears on line 3.

Figure 3 shows the structure summary of the routine report.

Figure 3
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 3, `$<ANONYMOUS#0>` on the first line refers to the 0th un-named VMS structure in the file, which has a size of 4 bytes and a field called `J` which is offset 0 bytes from the start of the structure. `$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.

Caller/callee routine cross-reference

The `CALLER/CALLEE ROUTINE CROSS-REFERENCE` section or the report (see Figure 4) lists each routine's subroutine, function or intrinsic callers and callees.

Figure 4
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 4 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; `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 6. 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 5 shows the differing members report for `BLK1` ordered by member names.

Figure 5
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

                .

                .

                .
```

The `SIZE =>` entry in Figure 5 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 Figure6) presents a numbered list of include files referenced.

Figure 6
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 6 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 7) lists subroutines, functions and common blocks contained in the report and the report page numbers on which they appear, ordered by object name.

Figure 7
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-referencer report page number on which it can be found.

Files

`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.

`fxref` support

`fxref`, the previous CONVEX FORTRAN cross-referencer, is no longer supported by CONVEX and will eventually be removed from the compiler. However, it can be invoked standalone or at compile time with the current compiler by specifying one of the following compiler options:

- `-xro`: invokes the old cross-referencer.
- `-xr1`: invokes the old cross-referencer using the single table sort option.
- `-sl`: produces the old cross-referencer source listing with line numbers.

For more information on `fxref`, see the `fxref(1f)` man page.

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, refer to the *CONVEX adb (Assembly-Language Debugger) User's Guide.*

Performance analyzer

The performance analyzer, CXpa, is an optional product that gathers and analyzes profiling data. The performance analyzer is an interactive tool that can monitor program activity at three levels:

- Routine
- Loop
- Block

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

The performance analyzer is described in detail in the *CONVEX Performance Analyzer (CXpa) User's Guide.* The following sections highlight some of the profiler features.

Routine-level profiling

Routine-level profiling produces summary information about the routines that are called during profiled execution of the program. This information includes the following:

- Total number of times the routine is called
- Total wall clock time in the routine and percentage of program total (minimum, maximum, average)
- Total CPU time in the routine and percentage of program total, including the cumulative time in called routines (minimum, maximum, average; the routines are sorted by the total CPU time)

- Net CPU time in the routine and percentage of program total, excluding the cumulative time in called routines (minimum, maximum, average)

Loop-level profiling

Loop-level profiling produces summary information about individual loops in the program. Certain loop optimizations affect how a loop is profiled. For example, if loop distribution, partial vectorization, or dynamic selection has been performed, each replicated copy of the loop is profiled separately.

The information provided for a loop includes the following:

- Starting line number of the loop within the source file
- Type of loop—scalar, vector, parallel, or parallel vector
- Number of times the loop is executed
- Vector length (minimum, maximum, average)
- Total CPU time in the loop, including the cumulative time in nested loops
- Chore count (minimum, maximum, average)
- Transformations performed by the compiler

Block-level profiling

Block-level profiling can be used to determine how many times each basic block in your code is executed. A basic block is a set of sequential assembly-language statements, the last of which changes the flow of control. Block-level profiling is separate from routine- and loop-level profiling. Programs to be profiled at the block level must be compiled using the `-pab` option.

Block-level profiling provides the following information for a routine:

- Starting line number within the source file
- Number of times the block is executed

CONVEX Consultant

The CONVEX Consultant is an optional product that includes a package of routines you can use to debug and analyze the performance of FORTRAN or C programs. The CONVEX Consultant includes the following:

- Symbolic debugger (`csd`)
- Postmortem dump (`pmd`)
- Profilers (`prof`, `bprof`, and `gprof`)

The *CONVEX Consultant User's Guide* describes each program and tells you how to use it effectively. The guide also discusses the proper techniques to follow when you are debugging optimized or parallelized code.

Symbolic debugger

The CONVEX symbolic debugger (`csd`) is a tool designed specifically for debugging FORTRAN and C programs running under the ConvexOS operating system. To generate the information necessary for using `csd`, compile your program with the `-db` option on the compiler command line.

The `csd` program enables you to:

- Debug a program at the source or assembly-language level
- Examine core dumps and obtain a symbolic runtime stack trace
- Debug programs containing multiple source modules
- Access program variables by name rather than by absolute address
- Debug optimized code

Note

The correspondence between source code lines and optimized code is lost because the optimizer may reorder lines. For this reason, using the debugger on code that is optimized may not be effective. Refer to Appendix B of the *CONVEX FORTRAN Language Reference Manual* for other limitations.

To support parallel processing, `csd` provides special debugging commands to assist in monitoring program execution on multiple processors. You can use these commands to determine how many threads exist, control the number of threads permitted, determine which thread is current, change the current thread, and examine the communication registers used by threads.

Postmortem dump

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 8. If you do not indicate a choice, the short-form dump is the default.

Table 8
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.

Profilers

The CONVEX Consultant package offers three different profilers that monitor the performance of your program:

- Standard profiler (`prof`)
- Basic block profiler (`bprof`)
- 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 9.

Table 9
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	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.

CXdb debugger

CONVEX CXdb is an optional window-based debugger that includes all the functionality of regular debuggers and is capable of debuging optimized code. 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.

Like the CONVEX Symbolic Debugger (provided with the CONVEX Consultant optional product), CXdb can perform these functions:

- 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
- Debug optimized code

In addition to these capabilities, CXdb provides the following functionality:

- 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 window (CXwindows only)
  - Examine (memory) window (CXwindows only)
  - Vector, scalar and communication register windows (CXwindows only)
  - Disassembly window (CXwindows only)
- Pulldown menus for novice users and expert menu buttons for fast access to important commands
- Capability to debug a running process
- Execution of debugger commands while your process is running
- Creation of aliases and macros to simplify debugger commands
- Capability to fully debug code that has been optimized through the `-O3` level
- Control stepping granularity--CXdb can step by units as small as expressions or assembly language instructions

- Set eventpoints that stop execution when the value of a variable changes, a signal is caught or an expression becomes true; eventpoint handlers can then execute a sequence of commands, which can include resuming execution
- Display two-dimensional arrays as a table, with correct row/column arrangement according to language-dependent array storage (unless the `C$DIR ROW_WISE` compiler directive is used; see the *CONVEX FORTRAN Optimization Guide* for details)
- Capability to access static program variables that are not visible from the current point of execution
- Capability to use debugger variables to store information without affecting program variables
- Capability to debug, edit and recompile a program without leaving CXdb
- Capability to configure CXdb to meet your debugging needs
- Access to the entire *CXdb Reference Manual online in a help window*

CXdb's windowing environment supports both line-oriented terminals and workstations capable of displaying CXwindows. This windowing environment eases the task of debugging programs that contain multiple threads of execution. Refer to *CONVEX CXdb Concepts* for additional information on CONVEX CXdb. Refer to Chapter 1, "Compiling Programs," for additional information on using the `-cxdb` command-`line` option with the CONVEX FORTRAN compiler.

`error` utility

The `error` utility takes the error messages produced by the compiler and disperses them back 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.

Example:

```
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" sectin in Chapter 1 of the *CONVEX FORTRAN User's Guide* for more information on `-LST`.

Calling conventions

5

Subprogram calls in CONVEX FORTRAN use the same calling conventions as that 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.

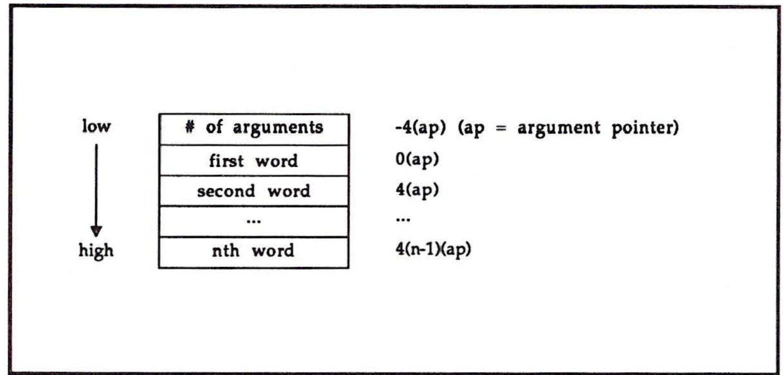
FORTRAN subprogram calling convention

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. A routine call involves passing a pointer to the list of arguments, called the *argument packet*.

FORTRAN argument packets

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 8 shows the layout of an argument packet in memory.

Figure 8
Argument packet: example 1



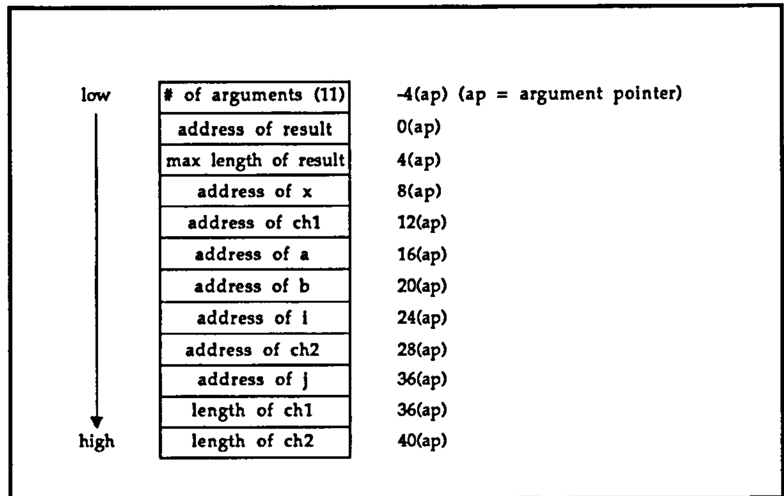
The first word is normally the address of the first argument, the second word is the address of the second argument, and so on. For character arguments, an extra by-value word containing the length of the character entity is added to the end of the list. For each character argument there is one extra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

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.

Figure 9 shows the argument packet layout resulting from 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 9
Argument Packet: example 2



Argument-passing mechanisms

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

- Passing arguments by immediate value, where the argument-packet entry is the value
- Passing arguments by reference, where the argument-packet entry is the address of the value

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.

%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 10 for details concerning values and the size of the argument.

Example:

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

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

Table 10
Built-in functions and defaults for argument lists

Data Type	Default	%REF	%VAL
Expressions:			
*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 10 for details concerning values and the size of the argument.

Function return values

The method of returning a value of a function depends on the data type of the value as summarized in Table 11.

Table 11
Function return values

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.

%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.

Example:

```
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 10 is an example of code you might write in assembler or C to call a FORTRAN subroutine. Refer to the *CONVEX Architecture Reference Manual* for information on the instruction set and calling procedures.

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. 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).

Figure 10
Calling a FORTRAN subroutine

```
CALL SUB (A,B,C)  !FORTRAN call with three real arguments

                         ! equivalent assembler calling sequence:
pushea   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 ap
pshea    3               ! push number of words in argument list
calls    _sub_           ! call the subroutine
ld.w     12(fp)a         ! restore ap
add.w    #16,sp          ! restore sp

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

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.

Procedure names

The compiler adds a leading and trailing underscore to the name of a common block or a FORTRAN subprogram to distinguish it from a C procedure or an external variable with the same user-assigned name. FORTRAN library procedure names have embedded underscores to avoid conflict with user-assigned subroutine names.

Data representations

Table 12 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.

Table 12
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;
REAL*16 *x*	struct {long long int upper,lower} x;
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. A COMPLEX or *DOUBLE COMPLEX* function is equivalent to a C routine with an additional initial argument pointing to where the return value is to be stored. See Table 13.

Table 13
Complex function: C equivalent

Complex FORTRAN function	Is equivalent to C function

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

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. See Table 14.

Table 14
Character functions: C equivalent

Character function...	Is equivalent to...	Can be invoked in C by...

`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

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 15.

Table 15
Character arguments: C equivalent

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

The following examples illustrate how to interface to FORTRAN from other languages and how to use argument-passing techniques.

Example 1

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)   ! IN FORTRAN
INTEGER I
REAL R
DOUBLE PRECISION D
D = I + R
END
```

C source code:

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

Assembly-language code:

```
sub: ld.w    @4(ap),s1 ; s0 = r
     ld.w    @0(ap),s0 ; s1 = i
     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

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:

```
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

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

FORTRAN source code:

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

Assembly-language code:

```
    .
    .
    .
ld.w      #0x000000a,s0      ; #3, 10
pshea     LC                 ; #3, ?LC
psh.w     s0                 ; #3
pshea     -10(fp)            ; #3, ?ch1
mov       sp,ap              ; #3
pshea     3                  ; #3
calls     _f_                ; #3, F
add.w     #0x0000010,sp      ; #3, 16
ldea      LU,a5              ; #3, A
ldea      -10(fp),a1         ; #3, ?ch1
ld.l      0x0(a1),s1
st.l      s1,0x0(a5)
ld.h      0x8(a1),s1
st.h      s1,0x8(a5)
```

Example 4

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

FORTRAN source code:

```
subroutine sub3
complex x,f
x = f (10)
end
```

Assembly-language code:

```
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_
ld.w    12(fp),ap    ; restore ap
add.w   #12,sp       ; restore sp
```

Example 5

This example shows how subprogram arguments are passed.

FORTRAN source code:

```
subroutine sub4
external f
call useit (f,x)
end
```

Assembly code:

```
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

This example shows how an array argument is passed.

FORTRAN source code:

```
subroutine sub5
real a(20)
call usearray (a,x,y)
end
```

Assembly 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
```

System utilities

6

The FORTRAN system utility routines provide a runtime interface between CONVEX FORTRAN programs and the ConvexOS operating system. The library in which the utility routines reside (/usr/lib/libU77.a) includes useful character and math functions. 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. Take, for example, the chdir function listed below and described in full in the chdir(3F) man page. (The manual 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
```

The routines described in this chapter that accept INTEGER*4 arguments must always be passed INTEGER*4 arguments, even if the -i1, -i2, -i8, -p8, or -pd8 option is set. Similarly, routines that accept REAL*4 arguments must be passed REAL*4 arguments regardless of which -r or -p option is set.

ConvexOS utilities

The calling sequences for the routines shown in Table 16 are also described in Section 3F of the man pages under the heading indicated in the Reference column of the table.

Table 16
Calling Sequences for ConvexOS Utilities

Name	Reference	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
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

Table 16
(continued)

Name	Reference	Description
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
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

Table 16
(continued)

Name	Reference	Description
stime	stime	return system time
system	system	execute a ConvexOS 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
unlink	unlink	remove a directory entry
wait	wait	wait for a process to end

The system utility

The system utility calls ConvexOS utilities that are not included in Table 16. The system utility executes a ConvexOS 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.

Example:

```
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 the 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 Section 3F of the online man pages.

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 16, or one of the functions listed in the following subsections.

date

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

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

idate

The `idate` utility returns the current month (*i*), day (*j*), and year (*k*).

```
SUBROUTINE idate(i,j,k)
INTEGER*4 i,j,k
```

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 ConvexOS 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 and is similar to the ConvexOS utility `rand`.

```
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 32.

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

Runtime errors and exceptions

7

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`.

During runtime, error or exception conditions can occur during I/O operations, from system-detected errors, invalid input data, arithmetic errors, or 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 to your needs

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 ConvexOS operating system.

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.

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 17 lists the name, number, and meaning of each runtime signal. Each signal has a default action associated with it. Except for SIGKILL and SIGSTOP, the signal utility allows this default action to be overridden.

Table 17
Signal names and numbers

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

* indicates that the default action for the signal is to end the program and produce a core dump
** indicates that the default action is to ignore the signal
*** indicates that the default action is to stop the program. The default action for all other signals is to end the program.

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 18 shows the mapping of exceptions to signals and codes.

Table 18
Mapping exceptions to signals and codes

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)
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)	

Error-processing utilities

This section describes general error-processing utilities you can use to attach your own signal handler, enable or disable certain arithmetic traps, and retrieve error numbers. The error-processing utilities 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` utilities 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 utilities 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` utilities 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`

The `errtrap` utility 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 utility 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 errors is set to OFF. Set them ON for debugging.

The argument to errtrap is produced by summing the appropriate flags from the following table:

Flag	Meaning	Default
'01'X	Trap integer overflow	OFF
'02'X	Trap floating underflow	OFF
'04'X	Trap intrinsic 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 errors.

The errtrap utility supersedes traper, which is maintained for upward compatibility.

Example:

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

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

Hardware differences

The `errtrap` utility 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 architectures. Table 19 enumerates these functions and their purpose. For more information refer to Chapter 8 of the *CONVEX Architecture Reference Manual.*

Table 19
Intrinsic instructions—C200, C3200, C3400, and C3800 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 11 shows the errors reported when the program is compiled and run on a CONVEX C1 series.

Figure 11
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 12 shows the errors reported when the program is compiled and run on a CONVEX C2 Series. Errors reported for different C2 or C3 Series architectures are similar.

Figure 12
C2 and C3 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. 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 utility designates a signal-handling routine. The signal utility has three parameters: the signal number, the condition handler, and a flag. The legal values of the flag are -1, 0, and 1.

Example 1:

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

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

Example 2:

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

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

Example 3:

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

Example 1:

Figure 13 shows the output of the traceback utility as the result of a floating point exception error.

Figure 13
`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]
```

Example 2:

Figure 14 shows the output of the `traceback` utility generated by a user-called subroutine.

Figure 14
`traceback` as 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` utilities

The `perror`, `gerror`, and `ierrno` utilities retrieve the system error message numbers. `perror` writes a message appropriate to the last detected system error to FORTRAN unit 0. `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.

Examples of signal handling

Figures 15 and 16 illustrate the use of the error processing utilities.

Figure 15
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 16
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
```

FORTRAN data representations

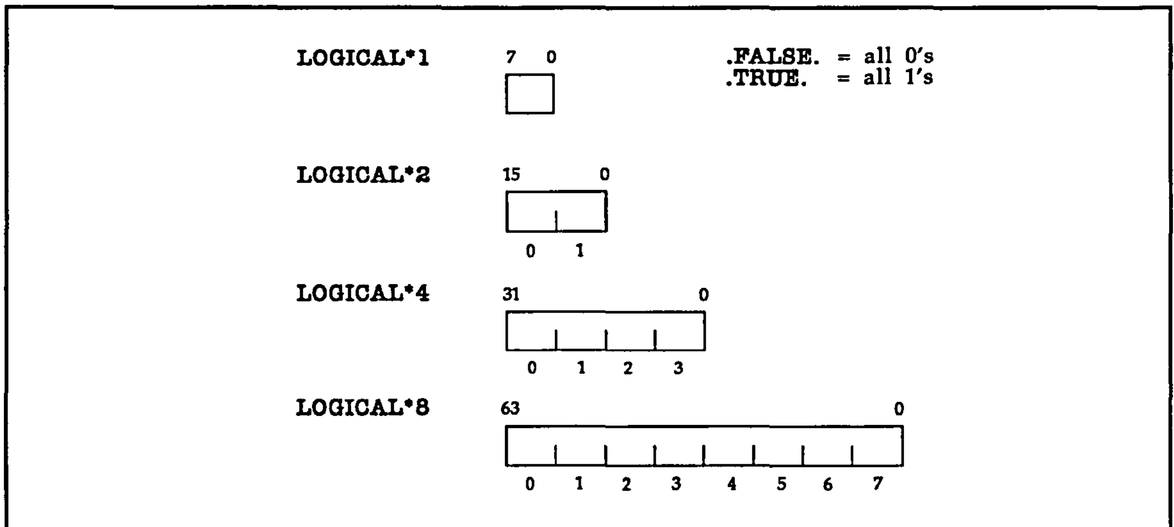
A

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 17.

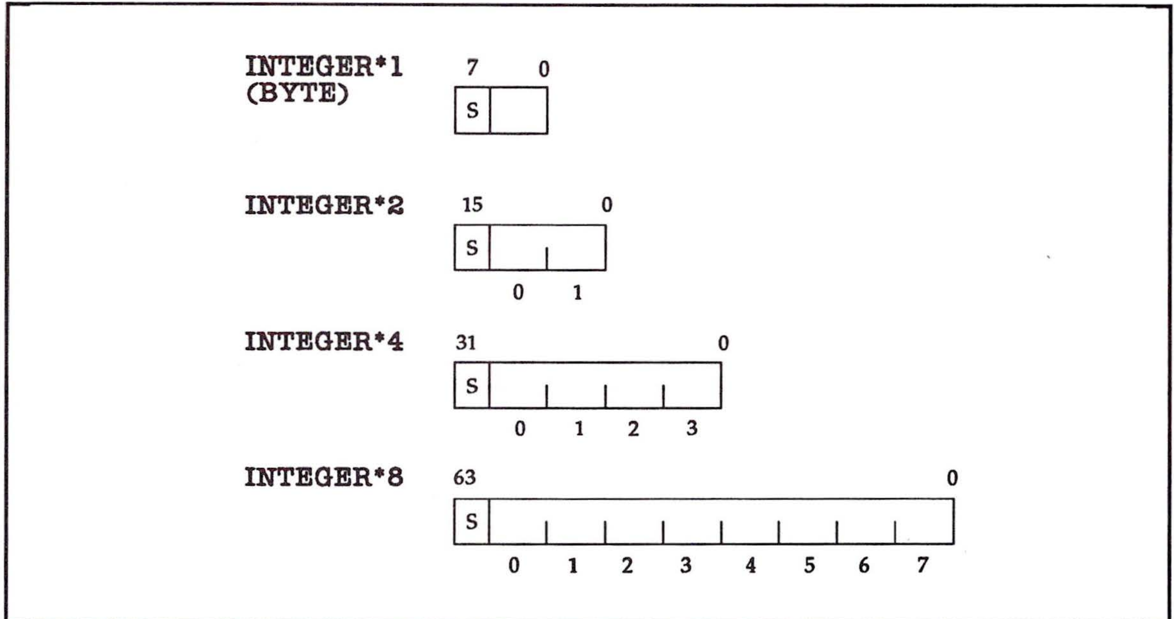
Figure 17
LOGICAL data type representation



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 18. INTEGER data types use the two's complement format.

Figure 18
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 -2^{63} to $+2^{63}-1$.

Real data representation

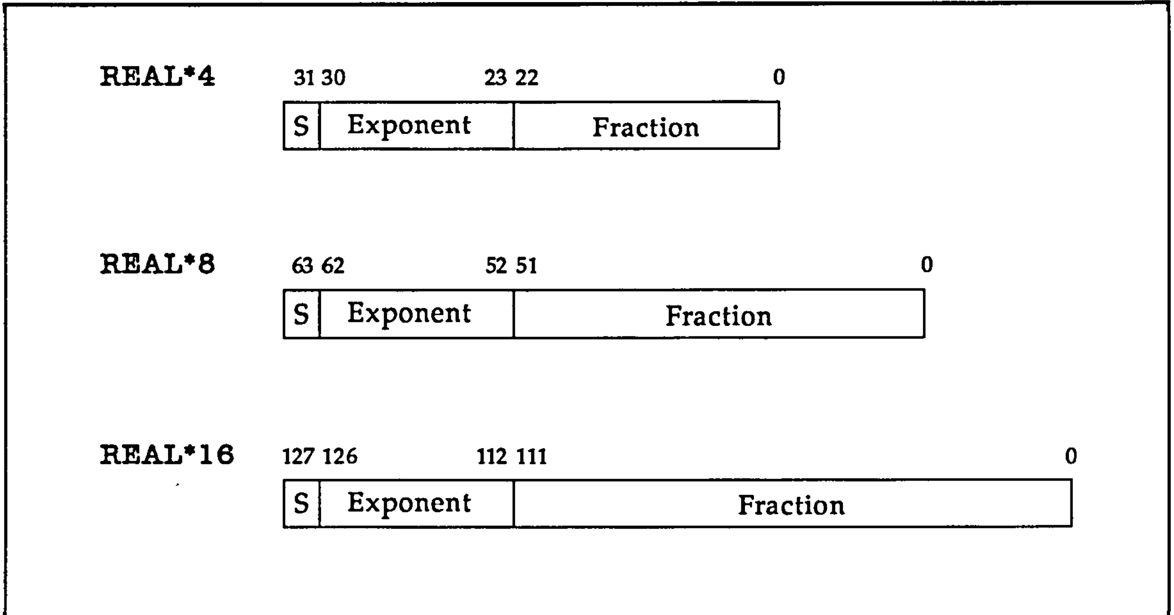
Floating-point variables are declared as REAL*4 (32 bits), REAL*8 (64 bits), or *REAL*16* (128 bits). REAL*4 and REAL*8 floating-point data can be represented either in native format or in IEEE format. To use IEEE format, your machine must be equipped with the IEEE support hardware. *REAL*16* data cannot be represented in IEEE format.

Note

The CONVEX hardware and runtimes only support the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

Figure 19 shows the internal representations of floating-point data. The positioning of the sign, exponent, and mantissa apply to native and IEEE formats.

Figure 19
REAL data type representation



The range of *REAL*16* data is

$$8.4052578577802337656566945433043 90 \times 10^{-4933}$$

through

$$5.9486574767861588254287966331400 34 \times 10^{+4931}$$

The ranges for REAL*4 and REAL*8 data are described in the following sections.

Native floating-point

The CONVEX native floating-point representation defines the following types of operands:

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:

$2.938740x10^{-39}$

through

$1.70141x10^{+38}$

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:

$5.56268464626801x10^{-309}$

through

$8.98846567431157x10^{+307}$

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 following types of operands:

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.

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

$1.797693134862312 \times 10^{+308}$

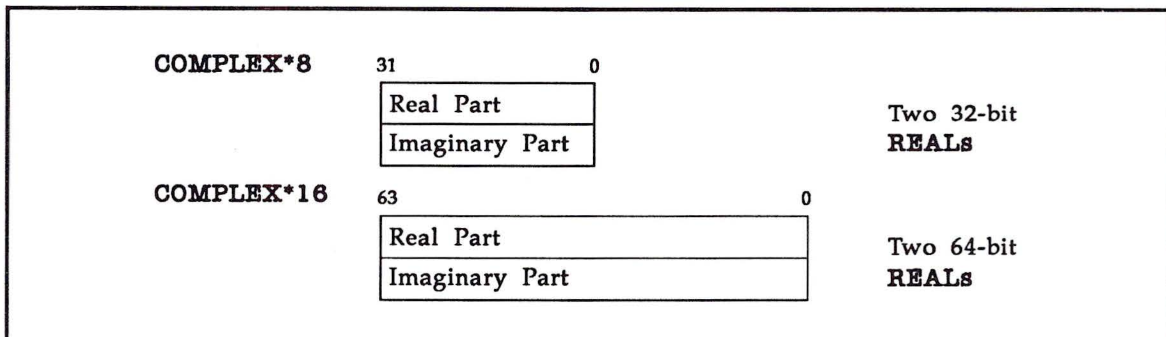
In the internal representation the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is

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 20.

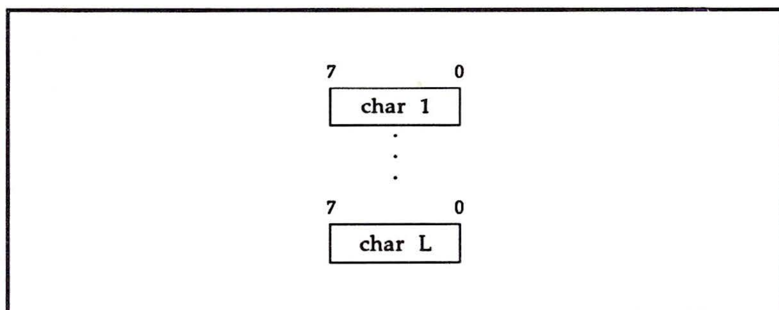
Figure 20
COMPLEX data type representation



Character representation

A character string is stored internally as a sequence of bytes, as shown in Figure 21. A character constant is limited to 4000 characters. Character strings formed at runtime can be of arbitrary length.

Figure 21
CHARACTER data type representation



Hollerith representation

A Hollerith constant is stored internally as a sequence of bytes and is limited to 2000 characters.

Compiler and runtime messages

B

The CONVEX FORTRAN compiler issues four kinds of diagnostic messages: error, warning, advisory, and vector summarization. All messages are output to `stderr`.

When the compiler has completed the syntactic and semantic analyses of a program, it aborts the compilation if user errors remain. An abort can also occur during optimization, for example, integer truncation during constant folding.

Compiler messages

The compiler message types are error, warning, advisory, and vector summarization. These messages appear on your screen unless you redirect them. 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.

Example:

```
fc file.f
```

sends compiler messages to the screen, while

```
fc file.f >&out
```

when executed under the `csh`, sends the messages to the file `out`. You can also use the `error` utility to insert diagnostic messages into your source file, where they appear as comments. This is a convenient way to locate the bugs while you are editing your source file.

Example:

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

When executed under the `csh`, this command 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`. You can write a simple `csh` 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.

Note

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

Runtime error messages

The runtime library reports errors encountered during execution. Runtime errors can be system-detected, arithmetic, or I/O errors. The runtime library provides default error processing and error message generation. All error messages are written to unit 0, `stderr`.

System errors

System errors can be returned either by the FORTRAN I/O library or by the FORTRAN utility library. In the former case, system errors are in the form of an I/O error message; in the latter case, the error number is returned as the value of the utility function (refer to section 3F of the man pages). The system errors generated by the operating system are described in the *ConvexOS Man Pages for Programmers* under the section `INTRO(2)`.

I/O errors generated by runtime library

The following system error messages are generated by the 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 `()`, 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 io not allowed`

The logical unit was opened for unformatted I/O.

`103 unformatted io not allowed`
The logical unit was opened for formatted I/O.

`104 direct io not allowed`
The logical unit was opened for sequential access, or the logical record length was specified as 0.

`105 sequential io 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 * 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 logical unit was not open.

`115 read unexpected character`
Certain format conversions cannot tolerate non-numeric 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 Technical Assistance Center (TAC).

`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`, etc., are checked for legitimacy. Often only nondefault forms are looked for.

`122 negative repeat count`

The repeat count for list-directed input must be a positive integer.

`123 illegal operation for unit`

`124 new record not allowed`

Encode and decode can only write and read single records.

`125 numeric keyword variable overflowed`

A keyword variable such as `ASSOCIATEVARIABLE` overflowed.

`126 record number is out of range`

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 files opened with the `READONLY` keyword.

`128 variable record format not allowed`

Direct-access files may not have variable-length records.

`129 record length exceeded`

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 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
Returned for direct-access reads when an attempt is made to read a record that does not exist.

135 reopening file with different unit not allowed

136 io list item type is incompatible with format code

137 unknown record length
A record length must be specified for the file.

138 asynchronous io not allowed on this file

139 synchronous io not allowed on this file

140 incompatible format structure - recompile
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 apparent recursive logical name definition

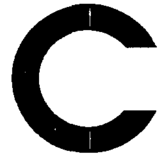
143 recursive input/output operations

144 out of free space, possibly from performing unformatted I/O

145 error in conversion of string to numeric

146 binary I/O data format conversion routine returned err
An error has been detected in the use of binary I/O.

Runtime libraries



This appendix describes the FORTRAN runtime libraries. Table 20 lists the names and contents of the standard FORTRAN runtime libraries.

Any libraries specified on the compiler command (`fc`) line with the `-l` loader option are searched before the standard libraries. The `/usr/lib/libI66.a` library is required only for FORTRAN 66 compatibility. This library is included automatically if the `-F66` compiler option is specified.

Note

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

The `libU77.a` library functions are also available. See the `intro(3f)` man page for a description of these functions.

Table 20
FORTRAN runtime libraries

Name	Contents
/usr/lib/libF77.a	Intrinsic function library
/usr/lib/libF77_p.a	Profiled intrinsic function library
/usr/lib/libI77.a	FORTRAN I/O library
/usr/lib/libI77_p.a	Profiled FORTRAN I/O library
/usr/lib/libU77.a	ConvexOS interface library
/usr/lib/libU77_p.a	Profiled ConvexOS interface library
/usr/lib/libU77_pa.a	CXpa profiled ConvexOS interface library*
/usr/lib/libU77p8.a	Equivalent to libU77.a except uses -p8 default variable lengths. See Chapter 1 of the *CONVEX FORTRAN User's Guide* for more information on the -p8 compiler option.
/usr/lib/libU77p8_p.a	Equivalent to libU77p8.a but for use with -p option.
.../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 the *CONVEX FORTRAN User's Guide* for more information on the -pd8 compiler option.
/usr/lib/libU77pd8_p.a	Equivalent to libU77pd8.a but for use with -p option.
.../palib/libU77pd8.a	CXpa profiled libU77pd8.a*
/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
/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

*CXpa is an optional product. CXpa profiled libraries may not be installed on your system or -pa libraries may be symbolically linked to the unprofiled versions. -pa libraries are present in the palib subdirectory of the directory containing the other libraries; this is normally /usr/lib.

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 by means of 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, for example, complex division, 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. Because vector arguments and results are passed in registers, vector lengths are restricted to a maximum of 128. This requires strip-mining by the compiler prior to a vector runtime routine call.

The functions in `libF77` 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

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 21.

Table 21
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

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 22 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 22
Intrinsic functions

Intrinsic	Runtime routine name	Arguments	Result
Square root			
SQRT	{for,mth}$r_sqrt	r	r
	mth$vr_sprt	vr	vr
DSQRT	{for,mth}$d_sqrt	d	d
	mth$vd_sqrt	vd	vd
CSQRT	{for,mth}$c_sqrt	c	c
	mth$vc_sqrt	vc	vc
CDSQRT	{for,mth}$z_sqrt	z	z
	mth$vz_sqrt	vz	vz
QSQRT	{for,mth}$q_sqrt	q	q
	mth$vq_sqrt	vq	vq
Natural logarithm			
LOG	{for,mth}$r_log	r	r
	mth$vr_log	vr	vr
DLOG	{for,mth}$d_log	d	d
	mth$vd_log	vd	vd
CLOG	{for,mth}$c_log	c	c
	mth$vc_log	vc	vc
CDLOG	{for,mth}$z_log	z	z
	mth$vz_log	vz	vz
QLOG	{for,mth}$q_log	q	q
	mth$vq_log	vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Common logarithm			
LOG10	{for,mth}$r_log10	r	r
	mth$vr_log10	vr	vr
DLOG10	{for,mth}$d_log10	d	d
	mth$vd_log10	vd	vd
QLOG10	{for,mth}$q_log10	q	q
	mth$vq_log10	vq	vq
Exponential			
EXP	{for,mth}$r_exp	r	r
	mth$vr_exp	vr	vr
DEXP	{for,mth}$d_exp	d	d
	mth$vd_exp	vd	vd
CEXP	{for,mth}$c_exp	c	c
	mth$vc_exp	vc	vc
CDEXP	{for,mth}$z_exp	z	z
	mth$vz_exp	vz	vz
QEXP	{for,mth}$q_exp	q	q
	mth$vq_exp	vq	vq
Sine			
SIN	{for,mth}$r_sin	r	r
	mth$vr_sin	vr	vr
DSIN	{for,mth}$d_sin	d	d
	mth$vd_sin	vd	vd
CSIN	{for,mth}$c_sin	c	c
	mth$vc_sin	vc	vc
CDSIN	{for,mth}$z_sin	z	z
	mth$vz_sin	vz	vz
QSIN	{for,mth}$q_sin	q	q
	mth$vq_sin	vq	vq
Sine (degree)			
SIND	{for,mth}$r_sind	r	r
	mth$vr_sind	vr	vr
DSIND	{for,mth}$d_sind	d	d
	mth$vd_sind	vd	vd
QSIND	{for,mth}$q_sind	q	q
	mth$vq_sind	vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Cosine			
COS	{for,mth}$r_cos	r	r
	mth$vr_cos	vr	vr
DCOS	{for,mth}$d_cos	d	d
	mth$vd_cos	vd	vd
CCOS	{for,mth}$c_cos	c	c
	mth$vc_cos	vc	vc
CDCOS	{for,mth}$z_cos	z	z
	mth$vz_cos	vz	vz
QCOS	{for,mth}$q_cos	q	q
	mth$vq_cos	vq	vq
Cosine (degree)			
COSD	{for,mth}$r_cosd	r	r
	mth$vr_cosd	vr	vr
DCOSD	{for,mth}$d_cosd	d	d
	mth$vd_cosd	vd	vd
QCOSD	{for,mth}$q_cosd	q	q
	mth$vq_cosd	vq	vq
Tangent			
TAN	{for,mth}$r_tan	r	r
	mth$vr_tan	vr	vr
DTAN	{for,mth}$d_tan	d	d
	mth$vd_tan	vd	vd
QTAN	{for,mth}$q_tan	q	q
	mth$vq_tan	vq	vq
Tangent (degree)			
TAND	{for,mth}$r_tand	r	r
	mth$vr_tand	vr	vr
DTAND	{for,mth}$d_tand	d	d
	mth$vd_tand	vd	vd
QTAND	{for,mth}$q_tand	q	q
	mth$vq_tand	vq	vq
Arc sine			
ASIN	{for,mth}$r_asin	r	r
	mth$vr_asin	vr	vr
DASIN	{for,mth}$d_asin	d	d
	mth$vd_asin	vd	vd
QASIN	{for,mth}$q_asin	q	q
	mth$vq_asin	vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Arc sine (degree)			
ASIND	{for,mth}$r_asind	r	r
	mth$vr_asind	vr	vr
DASIND	{for,mth}$d_asind	d	d
	mth$vd_asind	vd	vd
QASIND	{for,mth}$q_asind	q	q
	mth$vq_asind	vq	vq
Arc cosine			
ACOS	{for,mth}$r_acos	r	r
	mth$vr_acos	vr	vr
DACOS	{for,mth}$d_acos	d	d
	mth$vd_acos	vd	vd
QACOS	{for,mth}$q_acos	q	q
	mth$vq_acos	vq	vq
Arc cosine (degree)			
ACOSD	{for,mth}$r_acosd	r	r
	mth$vr_acosd	vr	vr
DACOSD	{for,mth}$d_acosd	d	d
	mth$vd_acosd	vd	vd
QACOSD	{for,mth}$q_acosd	q	q
	mth$vq_acosd	vq	vq
Arc tangent			
ATAN	{for,mth}$r_atan	r	r
	mth$vr_atan	vr	vr
DATAN	{for,mth}$d_atan	d	d
	mth$vd_atan	vd	vd
QATAN	{for,mth}$q_atan	q	q
	mth$vq_atan	vq	vq
Arc tangent with two arguments			
ATAN2	{for,mth}$r_atan2	r,r	r
	mth$vr_atan2	vr,vr	vr
	mth$vrr_atan2	vr,r	vr
	mth$rvr_atan2	r,vr	vr
DATAN2	{for,mth}$d_atan2	d,d	d
	mth$vd_ata2n	vd,vd	vd
	mth$vdd_atan2	vd,d	vd
	mth$dvd_atan2	d,vd	vd
QATAN2	{for,mth}$q_atan2	q,q	q
	mth$vq_atan2	vq,vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Arc tangent (degree)			
ATAND	{for,mth}$r_atand	r	r
	mth$vr_atand	vr	vr
DATAND	{for,mth}$d_atand	d	d
	mth$vd_atand	vd	vd
QATAND	{for,mth}$q_atand	q	q
	mth$vq_atand	vq	vq
Arc tangent with two arguments (degree)			
ATAN2D	{for,mth}$r_atan2d	r,r	r
	mth$vr_atan2d	vr,vr	vr
	mth$vrr_atan2d	vr,r	vr
	mth$rvr_atan2d	r,vr	vr
DATAN2D	{for,mth}$d_atan2d	d,d	d
	mth$vd_atan2d	vd,vd	vd
	mth$vdd_atand2	vd,d	vd
	mth$dvd_atand2	d,vd	vd
QATAN2D	{for,mth}$q_atan2d	q,q	q
	mth$vq_atan2d	vq,vq	vq
Hyperbolic sine			
SINH	{for,mth}$r_sinh	r	r
	mth$vr_sinh	vr	vr
DSINH	{for,mth}$d_sinh	d	d
	mth$vd_sinh	vd	vd
QSINH	{for,mth}$q_sinh	q	q
	mth$vq_sinh	vq	vq
Hyperbolic cosine			
COSH	{for,mth}$r_cosh	r	r
	mth$vr_cosh	vr	vr
DCOSH	{for,mth}$d_cosh	d	d
	mth$vd_cosh	vd	vd
QCOSH	{for,mth}$q_cosh	q	q
	mth$vq_cosh	vq	vq
Hyperbolic tangent			
TANH	{for,mth}$r_tanh	r	r
	mth$vr_tanh	vr	vr
DTANH	{for,mth}$d_tanh	d	d
	mth$vd_tanh	vd	vd
QTANH	{for,mth}$q_tanh	q	q
	mth$vq_tanh	vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	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 22
(continued)

Intrinsic	Runtime routine name	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}$r_int	r	r
	mth$vr_int	vr	vr
DINT	{for,mth}$d_int	d	d
	mth$vd_int	vd	vd
QINT	{for,mth}$q_int	q	q
	mth$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
Complex conjugate			
CONJG	for$c_conj	c	c
DCONJG	for$z_conj	z	z

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
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$vq_max	vq,vq	vq
Minimum (pairwise operation—not a reduction)			
IMIN0	mth$vi_min	vi,vi	vi
	mth$vii_min	vi,i	vi
JMIN0	mth$vj_min	vj,vj	vj
	mth$vjj_min	vj,j	vj
	for$j_imin	j,j	j
KMIN0	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 22
(continued)

Intrinsic	Runtime routine name	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 22
(continued)

Intrinsic	Runtime routine name	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}$k_mod	k,k	k
	mth$vk_mod	vk,vk	vk
	mth$vkk_mod	vk,k	vk
	mth$kvk_mod	k,vk	vk
AMOD	{for,mth}$r_mod	r,r	r
	mth$vr_mod	vr,vr	vr
	mth$vrr_mod	vr,r	vr
	mth$rvr_mod	r,vr	vr
DMOD	{for,mth}$d_mod	d,d	d
	mth$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 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Transfer of sign			
ISIGN	{for,mth}$i_sign	i,i	i
	mth$vi_sign	vi,vi	vi
	mth$vii_sign	vi,i	vi
	mth$ivi_sign	i,vi	vi
JISIGN	{for,mth}$j_sign	j,j	j
	mth$vj_sign	vj,vj	vj
	mth$vjj_sign	vj,j	vj
	mth$jvj_sign	j,vj	vj
KISIGN	{for,mth}$k_sign	k,k	k
	mth$vk_sign	vk,vk	vk
	mth$vkk_sign	vk,k	vk
	mth$kvk_sign	k,vk	vk
SIGN	{for,mth}$r_sign	r,r	r
	mth$vr_sign	vr,vr	vr
	mth$vrr_sign	vr,r	vr
	mth$rvr_sign	r,vr	vr
DSIGN	{for,mth}$d_sign	d,d	d
	mth$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}$i_and	i,i	i
JIAND	{for,mth}$j_and	j,j	j
KIAND	{for,mth}$k_and	k,k	k
Bitwise OR			
IIOR	{for,mth}$i_or	i,i	i
JIOR	{for,mth}$j_or	j,j	j
KIOR	{for,mth}$k_or	k,k	k
Bitwise XOR			
IIEOR	{for,mth}$i_xor	i,i	i
JIEOR	{for,mth}$j_xor	j,j	j
KIEOR	{for,mth}$k_xor	k,k	k
Bitwise complement			
INOT	{for,mth}$i_not	i,i	i
JNOT	{for,mth}$j_not	j,j	j
KNOT	{for,mth}$k_not	k,k	k

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Bitwise shift			
IISHFT	{for,mth}$i_shft	i,i	i
	mth$vi_shft	vi,vi	vi
	mth$ivi_shft	i,vi	vi
JISHFT	{for,mth}$j_shft	j,j	j
	mth$vj_shft	vj,vj	vj
	mth$jvj_shft	j,vj	vj
KISHFT	{for,mth}$k_shft	k,k	k
	mth$vk_shft	vk,vk	vk
	mth$kvk_shft	k,vk	vk
Bitwise extract			
IIBITS	{for,mth}$i_bits	i,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}$j_bits	j,j,j	j
	mth$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,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
Bitwise set			
IIBSET	{for,mth}$i_set	i,i	i
	mth$vi_set	vi,vi	vi
	mth$vii_set	vi,i	vi
	mth$ivi_set	i,vi	vi
JIBSET	{for,mth}$j_set	j,j	j
	mth$vj_set	vj,vj	vj
	mth$vjj_set	vj,j	vj
	mth$jvj_set	j,vj	vj
KIBSET	{for,mth}$k_set	k,k	k
	mth$vk_set	vk,vk	vk
	mth$vkk_set	vk,k	vk
	mth$kvk_set	k,vk	vk

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Bitwise test			
BITEST	{for,mth}$i_test	i,i	i
	mth$vi_test	vi,vi	vi
	mth$vii_test	vi,i	vi
	mth$ivi_test	i,vi	vi
BJTEST	{for,mth}$j_test	j,j	j
	mth$vj_test	vj,vj	vj
	mth$vjj_test	vj,j	vj
	mth$jvj_test	j,vj	vj
BKTEST	{for,mth}$k_test	k,k	k
	mth$vk_test	vk,vk	vk
	mth$vkk_test	vk,k	vk
	mth$kvk_test	k,vk	vk
Bitwise clear			
IIBCLR	{for,mth}$i_clr	i,i	i
	mth$vi_clr	vi,vi	vi
	mth$vii_clr	vi,i	vi
	mth$ivi_clr	i,vi	vi
JIBCLR	{for,mth}$j_clr	j,j	j
	mth$vj_clr	vj,vj	vj
	mth$vjj_clr	vj,j	vj
	mth$jvj_clr	j,vj	vj
KIBCLR	{for,mth}$k_clr	k,k	k
	mth$vk_clr	vk,vk	vk
	mth$vkk_clr	vk,k	vk
	mth$kvk_clr	k,vk	vk
Bitwise circular shift			
IISHFTC	{for,mth}$i_shftc	i,i,i	i
	mth$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,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}$k_shftc	k,k,k	k
	mth$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

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
String length			
LEN	for$s_len_j	s	j
String index			
INDEX	for$s_index_j	s	j
Character relationals			
LLT	for$s_lt_l	s	l
LLE	for$s_le_l	s	l
LGT	for$s_gt_l	s	l
LGE	for$s_ge_l	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
Absolute value			
IIABS	{for,mth}$i_abs	i	i
	mth$vi_abs	vi	vi
JIABS	{for,mth}$j_abs	j	j
	mth$vj_abs	vj	vj
KIABS	{for,mth}$k_abs	k	k
	mth$vk_abs	vk	vk
ABS	{for,mth}$r_abs	r	r
	mth$vr_abs	vr	vr
DABS	{for,mth}$d_abs	d	d
	mth$vd_abs	vd	vd
CABS	{for,mth}$c_abs_r	c	c
	mth$vc_abs_vr	vc	vc
CDABS	{for,mth}$z_abs_d	z	z
	mth$vz_abs_vd	vz	vz
QABS	{for,mth}$q_abs	q	q
	mth$vq_abs	vq	vq

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
Nearest integer			
ININT	{for,mth}$r_nint_i	r	i
	mth$vr_nint_vi	vr	vi
JNINT	{for,mth}$r_nint_j	r	j
	mth$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}$d_nint_j	d	j
	mth$vd_nint_vj	vd	vj
KIDNNT	{for,mth}$d_nint_k	d	k
	mth$vd_nint_vk	vd	vk
ANINT	{for,mth}$r_nint	r	r
	mth$vr_nint	vr	vr
DNINT	{for,mth}$d_nint	d	d
	mth$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}$q_nint	q	q
	mth$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
REAL*4 to REAL*8 conversion			
DBLE	{for,mth}$r_cvt_d	r	d
	mth$vr_cvt_vd	vr	vd

Table 22
(continued)

Intrinsic	Runtime routine name	Arguments	Result
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
Fortran 90 intrinsics			
DOT_PRODUCT	f90$dot_product	Varies†	Varies†
MATMUL	f90$matmul	Varies†	Varies†
ALL	f90$all	Varies†	Varies†
ANY	f90$any	Varies†	Varies†
COUNT	f90$count	Varies†	Varies†
MAXVAL	f90$maxval	Varies†	Varies†
MINVAL	f90$minval	Varies†	Varies†
PRODUCT	f90$product	Varies†	Varies†
SUM	f90$sum	Varies†	Varies†
MERGE	f90$merge	Varies†	Varies†
PACK	f90$pack	Varies†	Varies†
SPREAD	f90$spread	Varies†	Varies†
UNPACK	f90$unpack	Varies†	Varies†
TRANSPOSE	f90$transpose	Varies†	Varies†
MAXLOC	f90$maxloc	Varies†	Varies†
MINLOC	f90$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 23 perform exponentiation (**). The Arguments column lists the base type, followed by the exponent type.

Table 23
Exponentiation routines

Runtime routine name	Arguments	Result
mth$j_pow	j,j	j
mth$k_pow	k,k	k
mth$r_pow	r,r	r
mth$d_pow	d,d	d
mth$c_pow	c,c	c
mth$z_pow	z,z	z
mth$rj_pow_r	r,j	r
mth$dj_pow_d	d,j	d
mth$cj_pow_c	c,j	c
mth$zj_pow_z	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
{for,mth}$q_pow	q,q	q
mth$vq_pow	vq,vq	vq
{for,mth}$qj_pow_q	q,j	q
mth$vqj_pow_vq	vq,vj	vq

Complex programmed operators

The runtime routines listed in Table 24 perform complex multiplication and division. For division, the argument types are listed as "dividend, divisor."

Table 24
Complex programmed operators

Function	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

REAL*16 programmed operators

The runtime routines listed in Table 25 perform comparison and arithmetic functions.

Table 25
*REAL*16* programmed operators

Intrinsic	Runtime routine name	Arguments	Result
Comparison operators			
.LE.	{for,mth}$q_le_l	q,q	l
	mth$vq_le_l	vq,vq	vl
.LT.	{for,mth}$q_lt_l	q,q	l
	mth$vq_lt_l	vq,vq	vl
.GE.	{for,mth}$q_ge_l	q,q	l
	mth$vq_ge_l	vq,vq	vl
.GT.	{for,mth}$q_gt_l	q,q	l
	mth$vq_gt_l	vq,vq	vl
.EQ.	{for,mth}$q_eq_l	q,q	l
	mth$vq_eq_l	vq,vq	vl
.NE.	{for,mth}$q_ne_l	q,q	l
	mth$vq_ne_l	vq,vq	vl
Arithmetic Operators			
*	{for,mth}$q_mul	q	q
	mth$vq_mul	vq,vq	vq
/	{for,mth}$q_div	q	q
	mth$vq_div	vq,vq	vq
+	{for,mth}$q_add	q	q
	mth$vq_add	vq,vq	vq
-	{for,mth}$q_sub	q	q
	mth$vq_sub	vq,vq	vq
- (unary)	{for,mth}$q_neg	q	q

Vector mask programmed operators

The 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
```

String-manipulation programmed operators

The following are the string-manipulation programmed operators:

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_leq_l	String equal comparison
for$s_lne_l	String not equal comparison

Runtime routine data items

The runtime routines described below are associated with data values used by the runtime routine libraries and the FORTRAN compiler.

The label mth$vmones, which is two long words of all 1s, is used to load the vector mask register.

The values -2 through 130 are available in 6 data types:

mth$h_indx	*INTEGER*1*
mth$i_indx	*INTEGER*2*
mth$j_indx	INTEGER*4
mth$k_indx	*INTEGER*8*
mth$r_indx	REAL*4
mth$r_indx_i	REAL*4
mth$d_indx	REAL*8
mth$d_indx_i	REAL*8

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, `libI77.a`. 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`.

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 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

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`

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 the unit for I/O. The following operations are performed:

- If not currently open, opens file with default file attributes, for example, RECORDTYPE
- Initialize logical record buffer
- Compile runtime routine formats
- Save ERR and END flags
- Check for various error conditions, for example, illegal unit number, formatted I/O to file open for unformatted access

Entry points for 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 the 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 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 the I/O operation. This can require flushing the logical record buffer and completion of formatting, if any elements remain in the format string.

```
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 FORTRAN statements that they implement follows.

I/O runime routine	I/O statement
`for$back`	`BACKSPACE`
`for$close`	`CLOSE`
`for$end`	`ENDFILE`
`for$inqu`	`INQUIRE`
`for$open`	`OPEN`
`for$rew`	`REWIND`

IEEE Compatibility

D

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

- All finite element codes
- Computational chemistry codes requiring graphics pre- and postprocessing
- Libraries interfacing to workstation applications
- Graphics programs exchanging data with workstations

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). CONVEX processors do not, however, 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.

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.

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

CONVEX processors do not handle exceptions as specified by the IEEE standard, especially in the area of quiet and signaling `NaNs`.

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.

Part 2
CONVEX FORTRAN
Language Reference Manual

This part contains the *CONVEX FORTRAN Language Reference Manual* in its entirety. The information in this part provides a working definition of the CONVEX FORTRAN programming language and encompasses the American National Standard programming language FORTRAN, ANSI X3.9-1978. CONVEX FORTRAN extensions to the standard are covered in detail, as are supported Cray, VAX and Sun FORTRAN extensions. CONVEX FORTRAN's limited support of American National Standard programming language Fortran 90, ANSI X-3.198-199x, is also covered.

Introduction

1

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

Types of programs

An executable program consists of a main program and, optionally, one or more subprograms. A program unit is defined as a sequence of FORTRAN statements that ends with an `END` statement. *A program unit can also include an `OPTIONS` statement,* and comment lines.

A main program can begin with a `PROGRAM` statement but cannot begin with a `FUNCTION`, `SUBROUTINE`, or `BLOCK DATA` statement. A subprogram must begin with a `FUNCTION`, `SUBROUTINE`, or `BLOCK DATA` statement.

FORTRAN character set

The standard FORTRAN character set consists of the following:

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

The CONVEX extended character set includes the following:

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

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

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

Comment line

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

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

FORTRAN statements

FORTRAN statements are classified as executable or nonexecutable. Executable statements specify action; they form an execution sequence in an executable program. Nonexecutable

statements indicate characteristics, arrangement, and initial values of variables; contain editing information; classify program units; and designate entry points within subprograms.

If you are entering your CONVEX FORTRAN program from a terminal, you can enter statement lines of any length as long as you do not exceed the compiler's statement length limit (refer to Appendix D, System limits) or use the *-72* compiler option.

Executable statements include the following:

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

Nonexecutable statements include the following:

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

Each FORTRAN statement is divided into fields providing for statement label, continuation indicator, statement text, and a sequence number. Table 26 gives general rules for entering items into fields.

Table 26
FORTRAN fields

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

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

Character-per-column formatting

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

Statement label field

A statement label references statements in a program unit. Although any statement can have a label, only labeled executable statements and `FORMAT` statements can be referenced by other statements. Two statements in a program unit cannot have the same label.

The statement label, which must be a decimal integer, can be positioned in any column, 1 through 5. Blanks and leading zeros are ignored; for example, `5`, `05`, and `00005` are the same label.

Initial line

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

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

Continuation line

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

Statement text field

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

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

Debug statements

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

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

Compiler directives

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

ANSI-standard formatting

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

Tab-key formatting

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

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

Order of statements and lines

Figure 22 shows the order required for a CONVEX FORTRAN program unit. You can mix statements separated by vertical lines but not statements separated by horizontal lines. For example, you can intersperse `DATA` statements with statement function definitions and with executable statements. You cannot, however, mix statement function definitions with executable statements.

Figure 22
Required order of statements

<table>
<tr><td rowspan="7">Comment lines and *INCLUDE statements*</td><td colspan="4">*OPTIONS statement*</td></tr>
<tr><td colspan="4">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement</td></tr>
<tr><td rowspan="5">NAMELIST, FORMAT, and ENTRY statements</td><td colspan="3">IMPLICIT NONE statement</td></tr>
<tr><td colspan="2">IMPLICIT statements</td><td rowspan="2">PARAMETER statements</td></tr>
<tr><td rowspan="3">DATA statements</td><td>Other specification statements</td></tr>
<tr><td colspan="2">Statement function definitions</td></tr>
<tr><td colspan="2">Executable statements</td></tr>
<tr><td colspan="5">END statement</td></tr>
</table>

OPTIONS statement

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

The OPTIONS statement has the form

OPTIONS *option [option...]*

Table 27 summarizes the options that can be specified in the OPTIONS statement.

Table 27
OPTIONS statement

Option	Description
`-al`	Treat non-character arrays as assumed-size arrays.
`-cs`	Check subscript bounds at runtime.
`-ds`	Dynamic selection.
`-F66`	Select FORTRAN 66 rules.
`-i2 (i4, or i8)`	Integer variables not explicitly sized default to the user-specified size 2 (4, or 8).
`-na`	Suppress advisory messages.
`-no`	Perform no optimization.
`-nw`	Suppress warning messages.
`-On`	Perform optimization, where: `-O0` =basic block machine-independent scalar optimization `-O1` =`-O0` plus program unit machine-independent scalar optimization `-O2` =`-O1` plus vectorization `-O3` =`-O2` plus parallelization
`-r4` (or `r8`)	Real variables not explicitly sized default to the user-specified size 4 (or 8).
`-re`	Generate reentrant code.
`-rl`	Perform loop replication optimizations (`-ds` and `-ur` combined).
`-ur`	Loop unrolling.
`-sa`	Prevent compiler from generating precompiled argument packets.

Example:

```
OPTIONS -O2 -r8
```

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

INCLUDE statement

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

```
INCLUDE 'filename'
```

or

```
#include "filename"
```

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

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

Note

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

FORTRAN statement components

2

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

Symbolic names

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

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

Data types

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

Table 28 shows the data types available in CONVEX FORTRAN.

Table 28
Data types

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

† *REAL*16* is supported in native mode only.

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

For LOGICAL and INTEGER data types, the storage requirement can be controlled by the -i2, -i4, -i8, -p8 or -pd8 compiler options; the default is four bytes. For REAL and COMPLEX, the storage requirements can be controlled by the -r4, -r8, -p8, or - pd8 compiler options.

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

Table 29 shows the storage requirements for each data type. Defaults appear inside parentheses, but the defaults change when you use the -r*n*, -i*n*, - p8, -pd8, or -cfc compiler options. Refer to Chapter 1, "Compiling programs," of the *CONVEX FORTRAN User's Guide* for more information on compiler options.

Table 29
Storage requirements for data types

Data type	Storage requirements (bytes)
LOGICAL	*1, 2,* (4), *or 8*
*LOGICAL*1*	*1*
*LOGICAL*2*	*2*
LOGICAL*4	4
*LOGICAL*8*	*8*
INTEGER	*1, 2,* (4), *or 8*
*INTEGER*1, BYTE*	*1*
*INTEGER*2*	*2*
INTEGER*4	4
*INTEGER*8*	*8*
REAL	(4), *8, or 16*
REAL*4	4
REAL*8	8
*REAL*16*	*16*
COMPLEX	(8) *or 16*
COMPLEX*8	8
*COMPLEX*16*	16
DOUBLE PRECISION	(8) *or 16*
DOUBLE COMPLEX	(*16*)
CHARACTER	1
CHARACTER**len*	*len*
CHARACTER*(*)	

Conversion of data types

Where data types differ between the variable or array element on the left side and the expression on the right side of an assignment statement, CONVEX FORTRAN converts the expression on the right to the same data type as that on the left side. The compiler uses the following rules for conversion:

- *Treats* LOGICALS *as* INTEGERS *of the same length.*

- *Converts INTEGERS to longer types by sign extension and to shorter types by truncation. Truncation of significant bits causes an integer overflow.*
- Converts INTEGER values to REAL values by truncation. For example, 82762035 is too large to fit in a REAL*4 without loss of precision, so just enough rightmost bits of the binary representation are truncated to make it fit. After conversion, the value becomes 82762033.0.
- Converts REAL values to INTEGER values by truncation. Rounding is not performed. For example, `I = 5.9` assigns the value 5 to I.
- Converts a REAL value to a lower precision REAL value (for example, REAL*8 to REAL*4) by rounding to the lower precision.
- Converts a REAL value to a higher precision REAL value by zero-extending the mantissa. Converting a REAL value to a higher precision, however, does not increase the accuracy of the value.
- Converts COMPLEX values to other noncomplex numeric data types by converting the real part only. For example, `R = (15.6d0,7.5d6)` assigns the value 15.6 to R.
- Converts noncomplex values to COMPLEX by converting first to the appropriate precision REAL value to get the real part, and then assigning 0.0 or 0.0d0 to the imaginary part.
- Handles *COMPLEX-to-COMPLEX conversions by converting the real and imaginary parts separately, that is, as two REAL conversions.*

Note

Optimizing code containing type conversions (including implicit type conversions) can cause some code to vectorize poorly or not at all.

Constants

A constant is an arithmetic or logical value or a character string. It does not change during program execution. The form in which a constant is expressed determines the value and data type. The PARAMETER statement assigns a symbolic name to a constant. Refer to Chapter 3, "Specification statements," for more information on the PARAMETER statement.

Integer constants

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

Examples:

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

INTEGER constants take the default INTEGER precision. If the default precision is 2 or 4 bytes and the constant is too large, it is treated as an 8 byte constant; if it is too large to be represented in 8 bytes, the compiler truncates the constant to fit and issues a warning. The -in, -cfc, -p8, and -pd8 compiler options affect the default precision. Refer to Chapter 1, "Compiling programs," of the CONVEX FORTRAN User's Guide for more information.

Real constants

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

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

Examples:

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

Complex constants

Both `COMPLEX*8` and *`COMPLEX*16`* constants consist of a pair of real constants separated by a comma and enclosed in parentheses. The first constant is the real part and the second constant is the imaginary part. A `COMPLEX*8` constant is a pair of `INTEGER*2`, `INTEGER*4`, or `REAL*4` constants. *A `COMPLEX*16` constant is an ordered pair of integer, `REAL*4` or `REAL*8` constants.*

Example:

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

Octal constants

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

'cc...c'O

or

O'cc...c'

where c represents an octal digit.

Examples:

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

If you specify the `-vfc` *compiler option (refer to Appendix H, VAX FORTRAN compatibility), the compiler accepts octal constants of the form:*

```
"cc...c
```

where c represents an octal digit.

If you specify the `-cfc` *compiler option (refer to Appendix G, Cray FORTRAN compatibility), the compiler accepts octal constants of the form:*

```
cc...cB
```

where c represents an octal digit. This is compatible with Cray's Boolean octal constant form; refer to Appendix G, Cray FORTRAN compatibility, for details.

Hexadecimal constants

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

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

or

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

or

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

where c represents a hexadecimal digit.

Examples:

Valid	*Invalid*	*Reason*
`'1A6'X`	`'FFG'X`	*G not in range 0 - 9, A - F*
`'123'X`	`'12.4'X`	*Decimal point not allowed*
`'FFFFFFFF'X`	`1AB2X`	*No apostrophe*
`X'66F'`	`X129A'`	*Missing first apostrophe*
`'abc123ff'X`		

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

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

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

Example:

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

truncation/extension occurs as shown below:

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

Hollerith constants

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

*n*Hcc...c

where

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

c
: is a printable ASCII character.

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

Example:

Valid	Invalid	Reason
`4HHelp`	`0H`	Must contain at least one character

If you specify the `-cfc` compiler option (refer to Appendix G, Cray FORTRAN compatibility), the compiler accepts hollerith constants of the form:

*n*Lcc...c

where *n* and *c* take the same values as their standard-form counterparts.

The Cray form left justifies the constant in memory and zero-fills its storage space to the right. You must supply the the `-cfc` option when using this form. Refer to Appendix G, Cray FORTRAN compatibility, for more information.

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

- When a specific data type is required, that type is assumed for the constant.
- When the constant is used as an argument, no data type is assumed.

- *When the constant is used with a binary operator, the data type of the constant is that of the other operand. Although the data type is that of the other operand, the bit pattern is taken from the Hollerith constant.*
- *If you pass a Hollerith constant to a subroutine, you must pass it into a dummy argument of type* `INTEGER`, `REAL`, *or* `LOGICAL`. *Passing it into a* `CHARACTER` *variable will cause erroneous behavior. Refer to the ANSI FORTRAN 77 standard, page C-3, for more information.*
- *In any other context, the constant assumes an* `INTEGER*4` *data type, unless you change the default* `INTEGER` *length by using the* `-i2`, `-i8`, `-p8`, `-pd8` *or* `-cfc` *options.*
- *When a Hollerith constant continues across a line, you can use the* `-72` *option to imitate the behavior of other FORTRAN compilers (blank padding to column 72).*

Note

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

Logical constants

A logical constant represents the value true or false. The following forms are acceptable:

- `T` or `F`
- `.T. or .F.`
- `.TRUE. or .FALSE.`

Character constants

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

Examples:

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

You can delimit a character constant with quotation marks (") instead of apostrophes. In either case, the beginning delimiter must be the same as the ending delimiter. When quotation marks are the delimiters, use two consecutive quotation marks ("") within a string to represent one quotation mark.

Examples:

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

Note

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

Variables

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

Variable types are classified as LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or CHARACTER. This type describes the data's purpose, storage requirements and precision. It is given by the first character of the symbolic name if there is no type statement; variables starting with any letter I through N are implicitly typed INTEGER; all others are implicitly typed REAL.

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

Arrays

An array is a set of adjoining storage locations, with one to seven dimensions, identified by a single symbolic name. Individual storage locations are referenced with subscripts; *sections of the array can be specified within the subscripts, and the entire array can be referenced by its name.*

Conceptually, a one-dimensional array is a column of elements, with each element accessible by its subscript. A two-dimensional array can be thought of as a table of elements containing rows and columns, with each element accessible by a pair of subscripts. Refer to the "Array subscripts" section of this chapter for more information.

All the values in an array have the same data type and any value assigned is converted to the data type of the array. A `DATA` statement can be used to define an array element or an entire array before program execution. During execution, an array element is defined with an assignment or input statement; the entire array can be defined with an input statement, *or, through use of the `-f90` flag, with an assignment statement.*

The number of dimensions of an array is referred to as the array's rank. A rank definition has the form rank *n*, where *n* is the number of dimensions. Related to this is the array's shape, which describes the absolute number of elements for each dimension. Shape has the form (m_1, m_2, ... m_n), where *m* is the number of elements in the given dimension, and *n* is the rank of the array. Arrays having the same shape are said to be conformable. Scalar values are conformable with any shape array; when a scalar is used in this context, it can be thought of as filling an array of the proper shape with its value.

Rank, shape and conformability are important in discussing Fortran 90 array support, which CONVEX FORTRAN Version 7.0 provides in a limited capacity through use of the `-f90` compiler option. Supported Fortran 90 features are discussed in detail under applicable topics in this chapter.

Array declaration

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

Array declarations have the following form:

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

or

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

where

type
: is the type of the array elements. If the DIMENSION form is used, the array will take an implicit type based on its name if it is not explicitly typed elsewhere.

arrname
: is the name of the array.

lb
: is the lower dimension bound. Defaults to 1.

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

Examples:

```
      DIMENSION MYRAY(5)
C     ONE-DIMENSIONAL ARRAY WITH 5 ELEMENTS;
C     INTEGER TYPE BY NAME.

      COMMON RERAY(5,5)
C     TWO-DIMENSIONAL ARRAY WITH 25 ELEMENTS;
C     REAL TYPE BY NAME.

      INTEGER A(5,5,5)
C     THREE-DIMENSIONAL ARRAY WITH 125 ELEMENTS
C     (5 PLANES, 5 ROWS, 5 COLUMNS);INTEGER TYPE
C     STATEMENT OVERRIDES THE NAME RULE AND ANY
C     CONFLICTING IMPLICIT STATEMENT.

      CHARACTER*8 MRAY(2)
C     ONE-DIMENSIONAL CHARACTER ARRAY WITH 2
C     ELEMENTS, MRAY(1) AND MRAY(2); STORAGE
C     SPACE IS 8 BYTES FOR EACH ELEMENT.
```

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

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

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

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

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

Automatic arrays

The `-f90` and `-cfc` compiler options enable the use of automatic arrays. Any array declared local to a subroutine with one or more non-constant dimensions is considered an automatic array. Memory for automatic arrays is dynamically allocated on the stack on entry into the subroutine based on a variable dimension value passed into the subroutine (this is explained in detail further on). This stack space is freed on exit from the subroutine; automatic arrays cannot be saved using the `SAVE` statement.

Automatic arrays are declared with the following form:

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

or

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

where

type

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

arrname

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

n

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

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

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

```
SUBROUTINE SUB(I,J,K)
      .
      .
      .
DIMENSION IARR(I, J, 2*I)      !IMPLICITLY TYPED
INTEGER
REAL*8 X(J+I,K)
      .
      .
      .
```

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

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

Allocatable array declarations

Allocatable arrays are declared with the ALLOCATABLE statement. The array's rank must be supplied either in the ALLOCATABLE statement, in a DIMENSION statement, or in the

array's type declaration, which, if used, precedes the `ALLOCATABLE` *statement. Rank definitions have the following form:*

```
arrname(:[,:)
```

where arrname *is the name of the array, which is followed by parentheses containing one colon for each dimension of the array. Multiple colons are separated by commas. Code examples of this are given further on in this section. Note that the above example is not an executable statement, but rather a form for use within certain executable statements where a rank definition is required.*

Allocatable array declarations have the following form:

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

where

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

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

Example:

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

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

Array subscripts

A pair of parentheses that enclose one to seven subscript expressions separated by commas constitutes a subscript. A subscript specifies which array element is being referenced. It immediately follows the array name. Specify one subscript expression for each dimension defined for the array. For a two-dimensional array of `COMMON MYRAY(5,5)`, a valid reference is `MYRAY(2,4)`. A subscript expression can be any valid arithmetic expression, *or a Fortran 90 array section as described below.*

Fortran 90 allocatable arrays

Use of the `-f90` or `-cfc` flag enables Fortran 90 allocatable arrays. Declaration of these arrays is discussed under the "Allocatable array declarations" section of this chapter.

Storage for allocatable arrays is dynamically allocated on the heap at runtime when an `ALLOCATE` statement is executed. The heap space must be deallocated through use of the `DEALLOCATE` statement when the allocatable array is no longer needed.

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

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

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

Example:

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

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

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

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

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

Example:

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

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

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

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

Fortran 90 array sections

Use of the `-f90` compiler flag allows you to use array sections as operands in assignment statements and as arguments to many intrinsic functions as specified in Appendix A. An array section is a piece of an array bounded by specified elements in each dimension. The typical example of a rectangular section of a two-dimensional array is presented further on. An array section is denoted with the following form:

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

where

arrname
: is the array name.

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

j

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

step

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

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

Note:

Array section notation has similar syntax to rank definition, but they differ according to usage context.

Following is an example of how a rectangular section of a two-dimensional array is denoted.

```
Y = X(5:10,1:20:2)
```

This example assigns a section of the array `X` consisting of rows 5 through 10 and odd-numbered columns 1 through 20 to the array `Y` (a step of 2 starting at column 1 yields odd-numbered columns). This section has rank 2 and shape (6,10). `Y` must be declared as a two-dimensional array with exactly the same shape and rank as the section. Any time an array section is assigned to another array section, the array sections must have identical shape. Array sections of shape (1)—for example, `A(3:3,1:1)`—are still considered arrays and therefore are not accepted where a scalar variable is required. In other words, `A(3:3,1:1)` is not equivalent to `A(3,1)`; the former describes a single element array section and the latter describes a scalar value.

Note that Fortran 90 array section notation differs from array bound specifications described in the "Array declaration" section. In the context of an array declaration, the ":" operator is a bounds indicator, but in the context of an intrinsic argument or assignment statement operand, it is a section indicator.

Array storage

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

Fortran 90 array manipulation intrinsics

CONVEX FORTRAN Version 7.0 includes a subset of Fortran 90 array manipulation intrinsics which can be accessed through use of the `-f90` command line option. In CONVEX FORTRAN, Fortran 90 intrinsics cannot be nested. These intrinsics are explained in the following subsections.

Many Fortran 90 intrinsics provide for optional arguments. The American National Standard Programming Language Fortran 90, X3.198-199x, allows the keyword = argument syntax in the argument list so that these optional arguments can be skipped. CONVEX FORTRAN does not support this syntax. In CONVEX FORTRAN, if you want to omit an optional argument that is not the final argument in the argument list, you must supply a `0` in its place in the argument list. Optional arguments that fall at the end of the argument list do not require a place holder; they can be left out of the list.

Vector and matrix multiply functions

These routines are used to multiply vectors by vectors, matrices by matrices and vectors by matrices. A vector is defined as an array of rank one. A matrix is defined as an array of rank greater than or eqaul to one.

DOT_PRODUCT

`DOT_PRODUCT` performs dot-product multiplication of two numeric or logical vectors. It has the following form:

```
DOT_PRODUCT (vecta, vectb)
```

where

vecta

must be an array valued vector of numeric type (`INTEGER`, `REAL` or `COMPLEX`) or of type `LOGICAL`.

vectb

must be an array valued vector of numeric type if *vecta* is of numeric type or of type `LOGICAL` if *vecta* is of type `LOGICAL`. *vectb* must be the same size as *vecta*.

`DOT_PRODUCT` returns a scalar value. For numeric arguments, the type of this value is the same as the type of (*vecta* × *vectb*); for logical arguments, the result is the same as the type of (*vecta* `.AND.` *vectb*).

MATMUL

`MATMUL` performs matrix multiplication of two numeric or logical matrices. It has the following form:

```
MATMUL (mata, matb)
```

where

mata

must be an array valued rank one or rank two matrix of numeric type or type `LOGICAL`.

matb

must be an array valued rank one or rank two matrix of numeric type if *mata* is of numeric type or of type `LOGICAL` if *matb* is of type `LOGICAL`. If *mata* has rank one or two, *matb* must have rank two; if *matb* has rank one or two, *mata* must have rank two. The size of the first (or only) dimension of *matb* must equal the size of the last (or only) dimension of *mata*.

`MATMUL` returns an array with the same type as that of (*mata* × *matb*); for logical arguments, the result is the same as the type of (*mata* `.AND.` *matb*). The shape of the array is determined by the arguments:

- If *mata* has shape (*n*,*m*) and *matb* has shape (*m*,*k*), the result has shape (*n*,*k*).
- If *mata* has shape (*m*) and *matb* has shape (*m*,*k*), the result has shape (*k*).
- If *mata* has shape (*n*,*m*) and *matb* has shape (*m*), the result has shape (*n*).

Reduction functions

These functions perform various reduction operations on arrays. All reduction functions have at least one argument (an array) and can have one or two additional optional arguments.

In CONVEX FORTRAN, when a function with two optional arguments is called and the middle argument is to be omitted, the middle argument must be given a value of `0` *as a place holder.*

ALL

`ALL` determines whether all values of an array along an optional dimension are true. `ALL` has the following form:

```
ALL(mask[, dim])
```

where

mask
: an array of type `LOGICAL`.

dim (optional)
: must be a scalar of type `INTEGER` with a value in the range $1 \le dim \le n$, where *n* is the rank of *mask*. If *dim* is omitted, `ALL` is applied to the entire array and yields a scalar result.

`ALL` always yields a result of type `LOGICAL`. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank n-1 and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *mask*. In other words, the shape is identical to the shape of *mask*except it is missing the dimension specified in *dim*.

ANY

`ANY` determines whether any value in an array along an optional dimension is true. `ANY` has the following form:

```
ANY(mask[, dim])
```

where

mask
: an array of type `LOGICAL`.

dim (optional)
: must be a scalar of type `INTEGER` with a value in the range $1 \le dim \le n$, where *n* is the rank of *mask*. If *dim* is omitted, `ANY` is applied to the entire array and yields a scalar result.

ANY always yields a result of type LOGICAL. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of mask.

COUNT

COUNT counts the true elements in an array along an optional dimension. COUNT has the following form:

```
COUNT(mask[, dim])
```

where

mask
: an array of type LOGICAL.

dim (optional)
: must be a scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*. If *dim* is omitted, ANY is applied to the entire array and yields a scalar result.

COUNT always yields a result of type INTEGER. It is scalar if *dim* is omitted or if *mask* has rank one. Otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *mask*.

MAXVAL

MAXVAL returns the maximum value of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. MAXVAL has the following form:

```
MAXVAL(array[, dim[, mask]])
```

where

array
: an array of type INTEGER or REAL.

dim (optional)
: must be a scalar of type INTEGER with a value in the range $1 \leq dim \leq n$, where n is the rank of *mask*.

mask (optional)
: must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX FORTRAN, if you want to omit *dim* and supply *mask*, you must supply a `0` in *dim's* position.

`MAXVAL` yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank n-1 and of shape ($d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n$), where ($d_1, d_2, ..., d_n$) is the shape of *array*.

MINVAL

`MINVAL` returns the minimum value of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. `MINVAL` has the following form:

```
MINVAL(array[, dim[, mask]])
```

where

array
: an array of type `INTEGER` or `REAL`.

dim (optional)
: must be a scalar of type `INTEGER` with a value in the range ($1 \leq dim \leq n$), where *n* is the rank of *mask*.

mask (optional)
: must be an array of type `LOGICAL` that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX FORTRAN, if you want to omit *dim* and supply *mask*, you must supply a `0` in *dim's* position.

`MINVAL` yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank *n*-1 and of shape ($d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n$), where ($d_1, d_2, ..., d_n$) is the shape of *array*.

PRODUCT

`PRODUCT` returns the product of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. `PRODUCT` has the following form:

```
PRODUCT(array[, dim[, mask]])
```

where

array
: an array of type INTEGER, REAL or COMPLEX.

dim (optional)
: must be a scalar of type INTEGER with a value in the range $1 \le dim \le n$, where *n* is the rank of *mask*.

mask (optional)
: must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX FORTRAN, if you want to omit *dim* and supply *mask*, you must supply a 0 in *dim*'s position.

PRODUCT yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank n-1 and of shape $(d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *array*.

SUM

SUM returns the sum of the elements of an array along an optional dimension corresponding to the true elements of an optional mask. SUM has the following form:

```
SUM(array[, dim[, mask]])
```

where

array
: an array of type INTEGER or REAL.

dim (optional)
: must be a scalar of type INTEGER with a value in the range $1 \le dim \le n$, where *n* is the rank of *mask*.

mask (optional)
: must be an array of type LOGICAL that is conformable with *array*.

While both *dim* and *mask* are optional arguments, in CONVEX FORTRAN, if you want to omit *dim* and specify a *mask*, you must supply a 0 in *dim*'s position.

SUM yields a result of the same type as *array*. It is scalar if *dim* is omitted or if *array* has rank one. Otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.

Construction and manipulation functions

The functions listed in this section are used to construct or manipulate arrays of all types.

MERGE

MERGE constructs an array by merging two source arrays under a mask. MERGE has the following form:

```
MERGE(tsource, fsource, mask)
```

where

tsource
: is an array or scalar of any type. The elements in this array are masked by the .TRUE. elements of *mask*. If *tsource* is a scalar and *fsource* is an array, the value of *tsource* is broadcast to cover the shape of *fsource*.

fsource
: is an array or scalar of the same type as *tsource*. In CONVEX FORTRAN, *fsource* must be conformable with *tsource*. The elements in this array are masked by the .FALSE. elements of *mask*. If *fsource* is a scalar and *tsource* is an array, the value of *fsource* is broadcast to cover the shape of *tsource*.

mask
: is an array of type LOGICAL that represents a mask under which *tsource* and *fsource* are merged. *mask* must be conformable with *tsource*.

MERGE returns an array of the same type as *tsource*. The resulting array consists of elements of *tsource* that correspond to the .TRUE. elements of *mask* and elements of *fsource* that correspond to the other elements of *mask*. If *tsource* is an array, the result is of the same shape; if *tsource* is a scalar but *mask* is an array, the result is of the same shape as *mask*. If all three arguments are scalars, the result is *tsource* if *mask* is .TRUE., *fsource* otherwise.

PACK

Packs the elements of an array into an array of rank one under control of a mask. PACK has the following form:

```
PACK(array, mask, [vector])
```

where

array
: is an array of any type.

mask
: must be an array or scalar of type LOGICAL and must be conformable with *array*. The resulting array will contain only those elements of *array* that correspond to .TRUE. elements of *mask*.

vector (optional)
: is a rank one array of the same type as *array*, containing at least as many elements as there are .TRUE. elements in *mask*. If *mask* is a scalar, *vector* must contain at least as many elements as *array*.

PACK returns a rank one array with the same type as *array*. If *vector* is present, the resulting array is the same size as vector. If *vector* contains more elements than those masked out of *array*, the masked *array* elements fill vector consecutively from its beginning and any leftover *vector* elements are unchanged. If *vector* is omitted, the resulting array contains the same number of elements as there are true elements in *mask*, unless *mask* is a scalar with the value true, in which case the result size is the same as the size of *array*.

SPREAD

SPREAD replicates an array into a specified new dimension a specified number of times. SPREAD has the following form:

```
SPREAD(source, dim, ncopies)
```

where

source
: is an array or scalar of any type. It must have a rank of less than seven.

dim
: must be a scalar of type INTEGER with a value in the range $1 \le dim \le n+1$, where *n* is the rank of *source*. *dim* specifies the dimension to add.

ncopies

must be a scalar of type `INTEGER`. Specifies the number of copies of *source* to be made.

`SPREAD` returns an array of the same type as *source* with rank $n+1$, where n is the rank of source. If *source* is scalar, the shape of the result is (MAX(0,*ncopies*); if *source* is an array with shape $(d_1, d_2, ..., d_n)$, the shape of the result is $(d_1, d_2, ..., d_{dim-1}$, MAX(0, *ncopies*), $d_{dim}, ..., d_n)$

TRANSPOSE

`TRANSPOSE` returns the transpose of an array of rank two. `TRANSPOSE` has the following form:

```
TRANSPOSE (matrix)
```

where

matrix

is a matrix of any type. Must be of rank two.

`TRANSPOSE` returns a rank two array of the same type as *matrix* and with shape (m, n) where (n, m) is the shape of *matrix*. In other words, the rows and columns of *matrix* are swapped.

UNPACK

Unpacks a rank one array into an array under control of a mask. `UNPACK` has the following form:

```
UNPACK (vector, mask, field)
```

where

vector

is a rank one array of any type. It must have at least as many elements as there are `.TRUE.` elements in *mask*. *vector* contains the array to be unpacked.

mask

must be an array of type `LOGICAL`. *mask* provides the shape that *vector* will be unpacked into.

field

must be of the same type as *vector* and must be conformable with *mask*; scalar values are acceptable.

UNPACK returns an array of the same type as *vector* having the shape of *mask*. The positions in the result that correspond to the positions of the .TRUE. elements of *mask* are filled, in array element order, with the values from *vector*. All other elements of the result contain the value of *field* if *field* is a scalar, or corresponding elements of *field* if *field* is an array.

Location functions

These functions are used to locate the maximum and minimum values of the array.

MAXLOC

MAXLOC locates the maximum value in an array (along an optional mask if present). MAXLOC has the following form:

```
MAXLOC(array[, mask])
```

where

array
: must be an array of type INTEGER or REAL. Must not be a scalar.

mask (optional)
: must be a type LOGICAL array that is conformable with *array*.

MAXLOC returns a rank one array of type INTEGER and of size equal to the rank of *array*. The elements of the result are the subscripts of *array* which locate the maximum value of *array*. If this value occurs in more than one element of *array*, the location of the first occurrence is returned. If *mask* is present, the element returned is the maximum of those elements of *array* that correspond to true elements of *mask*.

MINLOC

MINLOC locates the minimum value in an array (along an optional mask if present). MINLOC has the following form:

```
MINLOC(array[, mask])
```

where

array
: must be an array of type INTEGER or REAL. Must not be a scalar.

mask (optional)
: must be a type LOGICAL array that is conformable with *array*.

`MINLOC` returns a rank one array of type `INTEGER` and of size equal to the rank of array. The elements of the result are the subscripts of array which locate the minimum value of array. If this value occurs in more than one element of array, the location of the first occurrence is returned. If mask is present, the element returned is the minimum of those elements of array that correspond to true elements of mask.

Expressions

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

Arithmetic expressions

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

The arithmetic operator specifies the computation to perform on the operands. This computation generates a numeric value. Arithmetic operators are listed in Table 30.

Table 30
Arithmetic operators

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

Operator precedence

When an expression has two or more operators and contains no parentheses, the operations are executed in the order given in Table 31.

Table 31
Arithmetic operator precedence

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

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

Examples:

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

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

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

Data type priority

Where operands of different data types are combined in an arithmetic expression, the higher-ranked argument determines the type, and the greater precision argument determines the precision. Data type ordering is shown in Table 32.

Table 32
Data type priority

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

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

The data types of arithmetic expressions follow certain conventions:

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

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

Relational expressions

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

Logical expressions

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

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

The evaluation of a logical expression that has two or more logical operators is based on operator precedence as shown in Table 33.

Table 33
Logical operator precedence

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

† Evaluation of the .OR. and .AND. operators is short-circuited. Refer to Chapter 6 for mor information.

The logical operator .XOR. is the same as .NEQV. Operators on the same level of precedence are interpreted from left to right. Arithmetic rules for operator precedence apply for evaluation. Mixed operations are evaluated first by arithmetic rules of precedence, next by relational operations, and last by logical operations. Logical operands appearing in IF conditionals are short circuited; refer to the "Short circuit evaluation of conditionals" section of Chapter 6, Control statements.

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

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

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

Character expressions

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

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

Examples:

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

Character substrings

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

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

where *var* is a character variable or array name, *subscript* is an array subscript (required only when *var* is an array), *expr1* is an optional numeric expression indicating the leftmost character

position of the substring, and *expr2* is an optional numeric expression indicating the rightmost character position of the substring.

If *expr1* is omitted, a value of 1 is assumed; if *expr2* is omitted, the length of the character variable is assumed. The value of *expr1* must be positive and less than or equal to *expr2*. The value of *expr2* must be less than or equal to the length of the string.

Example:

If

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

then

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

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

Example:

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

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

Constant expressions

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

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

- *In the compile-time logical expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or one of the functions* `IAND`, `IOR`, `NOT`, `IEOR`, `LGE`, `LGT`, `LLE`, *or* `LLT`, *with constant operands.*
- *In the compile-time character expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or the function* `CHAR` *with a constant operand.*
- *In the compile-time arithmetic expression, each operand is a constant, the symbolic name of a constant, another compile-time constant expression, or one of the functions* `MIN`, `MAX`, `ABS`, `MOD`, `ICHAR`, `DIM`, `DPROD`, *or* `IMAG` *with constant operands.*

Specification statements

3

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

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

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

COMMON statement

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

The COMMON statement has the form:

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

where

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

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

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

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

CONVEX FORTRAN supports Cray TASK COMMON statements when the -cfc flag is specified. Refer to Appendix G, Cray FORTRAN compatibility, for more information.

`IMPLICIT` statement

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

```
IMPLICIT typ (a[,a]...) [,[typ] (a[,a]...)]...
```

or

```
IMPLICIT NONE
```

where

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

a
: is one letter or a range of letters; the range is expressed as first letter of range, minus sign (-), last letter of range (for example, `F-H`).

len
: is an optional length specifier for the data type.

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

The `IMPLICIT NONE` form of the statement overrides all implicit defaults except intrinsic function types. When using `IMPLICIT NONE`, you must explicitly declare the data types of all symbolic names in the program unit. If you specify `IMPLICIT NONE`, no other `IMPLICIT` statement can be included in the program unit.

Examples:

```
      IMPLICIT COMPLEX(F,H-J)
C     ANY NAME BEGINNING WITH THE LETTER F OR
C     ANY OF THE LETTERS H, I, J IS TYPE COMPLEX

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

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

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

Type-declaration statements

Type statements are numeric, logical, or CHARACTER type-declarations. Type statements override the name rule and IMPLICIT statements. You can also use these statements to specify array dimensions.

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

The following subsections include examples of type-declaration statements.

Numeric type-declaration statements

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

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

where

type
: is any data type specifier except CHARACTER.

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

clist
: *is a list of constants. (Refer to the "Data statement form" section of Chapter 4.)*

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

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

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

CHARACTER type-declaration statements

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

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

where

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

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

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

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

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

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

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

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

DIMENSION statement

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

Example:

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

The preceding example specifies a type INTEGER two-dimensional array with 20 elements. The product of the dimension declarators, (4,5), determines the total number of storage elements assigned to the array. When you specify two or more array declarators, separate them by commas. Note that upper and lower dimension bounds can be used here, as described in the "Array declaration" section of Chapter 2.

EQUIVALENCE statement

The EQUIVALENCE statement causes two or more entities within a program unit to refer to the same storage area. Thus, the same storage unit can be referenced by more than one name. Each statement contains two or more variables, array elements, array names, or substring names, separated by a comma. An array must be dimensioned with a DIMENSION, type, or COMMON

statement before it or any of its elements can be equivalenced. All elements contained in the same set of parentheses are allotted storage in the same location.

Example:

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

In this example, variables F, G, and H share the same location; the 9th element in array MYARRAY and the 10th element in array COM share the same location; the variables L, M, and N share the same location.

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

Note

Use of the EQUIVALENCE statement can interfere with program optimization.

Equivalencing arrays

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

Example:

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

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

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

or

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

also align the two arrays in the same manner as `EQUIVALENCE (A(2),B(2,2))`. Table 34 shows how these statements align the arrays.

Table 34
Array locations

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

Two or more elements of the same array cannot share the same storage location. For example, the following statements are invalid because they allocate the same storage for `A(1)` and `A(3)`.

Example:

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

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

Example:

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

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

Equivalencing substrings

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

Example:

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

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

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

Using EQUIVALENCE in common blocks

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

Example:

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

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

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

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

PARAMETER statement

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

Standard PARAMETER statement

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

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

Examples:

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

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

Alternate PARAMETER statement

An alternate form of the `PARAMETER` statement is available. This `PARAMETER` statement also assigns a symbolic name to a constant, but its list is not bounded by parentheses and the form

of the constant determines the data type of the variable. The alternate PARAMETER *statement does not conform to the ANSI standard but has the following form:*

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

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

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

The symbolic name of a constant assumes the data type of its corresponding expression. Therefore you cannot specify the data type of a parameter in a type-declaration statement, and the initial letter of the constant name does not affect the data type.

Example:

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

To use the alternate PARAMETER *statement with only one constant, specify the* -vfc *compiler option. (Refer to Appendix H, "VAX FORTRAN compatibility".)*

PROGRAM statement

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

The PROGRAM statement has the form:

```
PROGRAM pgm
```

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

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

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

NAMELIST statement

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

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

where

nlgrpname
: is a symbolic name representing the list of entities to be read or written.

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

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

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

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

The following statement illustrates the use of NAMELIST:

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

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

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

EXTERNAL statement

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

```
EXTERNAL n [,n]...
```

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

If an EXTERNAL statement declares an intrinsic name as an external procedure, all references to the intrinsic name are treated as references to an external procedure rather than as calls to an intrinsic function. For example, if you declare COS in the EXTERNAL statement (EXTERNAL COS), all subsequent references are to an external procedure COS, not the intrinsic function COS.

INTRINSIC statement

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

```
INTRINSIC n [,n]...
```

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

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

SAVE statement

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

The SAVE statement has the following format:

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

where *n* is a variable name, staticly-sized array name, or a named common block contained between slashes (for example, /NCOM/). Dummy argument names, subprogram or function names, automatic or allocatable array names, or common block entity names are not allowed. When a common block name appears in a SAVE statement, all the variables and arrays in the common block are saved.

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

ALLOCATABLE statement

Allocatable arrays are dynamic arrays of predetermined rank. Storage for allocatable arrays is allocated on the heap at runtime when an ALLOCATE *statement is executed, and must be deallocated through use of the* DEALLOCATE *statement when the array is no longer needed. Refer to Chapter 2, "FORTRAN statement components", for more information on the* ALLOCATE *and* DEALLOCATE *statements.*

The ALLOCATABLE *statement is used to declare allocatable arrays. You must supply either the* `-f90` *or* `-cfc` *compiler options if you declare allocatable arrays in your program. Allocatable array declarations have the following form:*

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

where

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

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

An allocatable array's rank must be supplied either in the ALLOCATABLE *statement, in a* DIMENSION *statement, or in the array's type declaration, which, if used, precedes the* ALLOCATABLE *statement. Rank definitions are discussed in detail in the "Allocatable array declarations" section of Chapter 2, "FORTRAN statement components".*

DATA statement

4

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

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

DATA statement form

The DATA statement has the following form:

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

where

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

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

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

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

You can repeat the same value for more than one element by placing a nonzero, unsigned integer constant indicating the number of repetitions and an asterisk (*) before the value.

Example:

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

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

The previous example can also be represented as:

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

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

Example:

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

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

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

Example:

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

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

Implied-DO

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

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

where

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

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

m_1, m_2, m_3
: are integer constant expressions that can contain implied-DO variables of other implied-DO lists. The constants specify the initial value, terminal value, and increment, respectively, for the integer variable. If you omit the comma and the value of m_3, the increment value defaults to 1. The increment count must be positive.

Examples using implied-DO:

The statements:

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

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

The statements:

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

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

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

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

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

DATA statement extensions

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

- *If an octal or hexadecimal constant is assigned to a variable or array element, the data type of the variable or array element determines the number of digits that can be assigned. Where the constant has fewer digits than the variable or array element, the constant is extended on the left with zeros. If the constant has more digits than can be stored, the constant is truncated on the left.*
- *If a Hollerith or character constant is assigned to a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the variable or array element. Where the Hollerith or character constant has fewer characters than the variables or array element, spaces are added to the right of the constant. If the constant has more characters than can be stored, excess characters on the right of the constant are eliminated.*

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

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

Following these restrictions permits the entity to be initialized with the character that has the ASCII code specified by the constant, which, in turn, allows a character entity to be initialized to any 8-bit ASCII code.

*The next example initializes the real array R to all zeros, the real variable P, the character variable C*4 to 'TEST', and the character variable CR*1 to the ASCII character code '0D'X.*

Example:

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

Arrays and variables in COMMON can be initialized in DATA statements in program units other than BLOCK DATA subprograms. Each variable or array element can only be initialized once.

Assignment statements 5

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

The assignment statement has the form:

v = *ex*

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

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

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

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

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

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

Character conversions

The character assignment statement has the following form:

$v = ce$

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

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

Example:

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

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

Fortran 90 array assignments

The CONVEX FORTRAN Version 7.0 compiler includes limited support of Fortran 90 array manipulation features. These features can be accessed using the `-f90` compiler option on the `fc` command line. Refer to Chapter 1, "Compiling programs," of the CONVEX FORTRAN User's Guide for more information on the `-f90` option.

The ability to use array-valued expressions in assignment statements is one Fortran 90 feature supported by CONVEX FORTRAN Version 7.0. This feature allows you to assign a value or expression to an entire array or array section with one assignment, using the following form:

$arr = ex$

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

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

Example:

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

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

Array sections can be similarly assigned. Refer to the section "Fortran 90 array sections" in Chapter 2 for more information on array sections.

Data conversion rules

Table 35 summarizes the data conversion rules for assignment statements. *These rules apply for both arrays and scalar variables of the indicated type.*

Table 35
Conversion of expressions

Type of variable (V)	Type of expression (E)	Value assigned
INTEGER/ LOGICAL	INTEGER/ LOGICAL	Direct assignment of E to V.
	REAL	Truncate E to INTEGER and assign to V.
	REAL*8	Truncate E to INTEGER and assign to V.
	*REAL*16*	*Truncate E to INTEGER and assign to V.*
	COMPLEX	Truncate real part of E to INTEGER and assign to V. Imaginary part not used.
	*COMPLEX*16*	*Truncate real part of E to INTEGER and assign to V. Imaginary part not used.*

Table 35
(continued)

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

Table 35
(continued)

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

ASSIGN statement

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

```
ASSIGN s TO i
```

where

s

is the label of an executable statement or `FORMAT` statement in the current program unit and

i

is an `INTEGER` variable name.

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

Example:

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

The above example transfers control to a statement with the label `75`. Do not use the `ASSIGN` statement for arithmetic purposes. For example, `ASSIGN 75 TO M` is not equivalent to `M = 75`. If you define the integer variable with a statement label, you can redefine it with the same or a different statement label value or integer value. For example, the statement

```
M = 50
```

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

Control statements

6

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

- GOTO—Unconditional, computed and assigned
- IF—Arithmetic, logical and block, including ELSE IF, ELSE and END IF
- DO—Indexed DO and *DO WHILE*
- *END DO*
- CONTINUE
- CALL
- RETURN
- STOP
- PAUSE
- END

GOTO statements

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

Unconditional GOTO statement

The unconditional GOTO statement has the form:

GOTO *sl*

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

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

Example:

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

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

Computed GOTO statement

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

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

where

slist
: is a list of labels of executable statements within the current program unit separated by commas (l_1, l_2, ...).

e
: is an arithmetic expression.

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

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

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

As a CONVEX extension, the arithmetic expression (e) is converted to an integer data type when necessary.

Example:

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

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

Assigned GOTO statement

The assigned GOTO statement has the following form:

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

where

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

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

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

Example:

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

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

IF statements

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

Arithmetic IF statement

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

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

where

e
: is an arithmetic expression.

l_1, l_2, l_3
: are labels of executable statements contained within the current program unit. All three labels must be included. Do not use labels of statements within DO statement blocks or IF statement blocks.

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

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

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

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

Example:

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

Logical IF statement

The logical IF statement has the form:

```
IF (le) es
```

where

le

is a logical expression.

es

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

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

Consider the following example.

Example:

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

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

Block IF statement

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

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

```
IF (le) THEN
   .
   .
   .
END IF
```

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

Consider the following example:

Example:

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

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

Another variation of the block IF statement has the following form:

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

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

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

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

A more complex block IF statement has the form:

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

This IF block allows any number of additional logical expressions (*le*) to be specified. If the value of the first logical expression (le_1) is false, the ELSE IF expressions are evaluated until the value of the expression is true. If le_2 through le_n are false, those block sequences are skipped and control transfers to the optional ending ELSE statement, or, if none is present, to the END IF statement. The next example contains an IF block with ELSE IF THEN and ELSE statements.

Example:

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

The IF block is evaluated sequentially. Evaluation of each ELSE IF THEN statement continues until a true value is determined. Then the statements associated with that ELSE IF THEN statement execute and control transfers to the END IF statement, which transfers control to the next executable statement. If all ELSE IF THEN statements evaluate to false, the ELSE statement block, if there is one, executes.

Nested block IF statements

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

Example:

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

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

Short-circuit evaluation of conditionals

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

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

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

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

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

*Short-circuit evaluation works with all types of IF statements (arithmetic, logical, and block). Performing arithmetic (+, –, *, /) on or applying non-logical operands or functions to a logical expression disables short circuit evaluation within that expression. Logical valued expressions used as arguments to function calls within an IF statement's conditional expression are not short-circuited. Note that the binary operators .EQ., .NE., .LE., .GT., and .GE. always produce a logical result.*

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

DO statement

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

```
DO [sl [,] ] v = ex1, ex2[, ex3]
```

where

sl

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

The label identifies the last statement (terminal statement) of the DO loop and the label must textually follow the DO statement. You cannot use an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, END, or DO statement as the terminal statement.

v

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

ex_1

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

ex_2

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

ex_3

is an integer, real, double-precision, or quad-precision expression that indicates the increment for *v* after each time the body of the DO loop is executed. If you omit the increment value, it defaults to 1.

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

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

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

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

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

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

Nested DO loops

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

Example:

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

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

Extended-range DO loops

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

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

- *You can transfer into the range of a DO statement only if you make the transfer from the extended range of the same DO statement; the transfer is invalid otherwise.*

- *The extended range of a* DO *statement must not change its control variable.*

The following example illustrates valid control transfer into the range of a DO *statement from its extended range.*

Example:

```
5    DO 10 I = 1,10
     IF(A(I).LE.0)GOTO 20          !DO LOOP
30   A(I) = -1
10   CONTINUE
     RETURN
20   A(I) = F(X(I))                !EXTENDED RANGE
     GOTO 30
```

The following example illustrates invalid transfer into the range of a DO *statement.*

Example:

```
     IF (FLAG)GOTO 5
     I = 3
     GOTO 30 !INVALID CONTROL TRANSFER TO DO LOOP
5    DO 10 I = 1,10
     IF (A(I).LE.0)GOTO 20
30   A(I) = -1
10   CONTINUE
20   RETURN
```

DO WHILE statement

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

```
DO [s [,] ] WHILE (e)
```

where

s

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

e

is a logical expression.

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

Example:

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

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

END DO statement

The END DO statement ends the DO and DO WHILE statements. You must include the END DO statement at the end of a DO loop if the DO or DO WHILE statement defining the loop does not contain a terminal-statement label. You can also use the END DO statement as a labeled terminal statement if the DO or DO WHILE statement does contain a terminal-statement label.

Example:

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

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

CONTINUE statement

Because the execution of a CONTINUE statement has no effect, you can place it anywhere in a program that an executable statement is allowed. When used as a terminal statement in a DO loop, a CONTINUE statement must be labeled; otherwise, no label is required. This statement has the following form:

```
[S] CONTINUE
```

where S is an optional statement label.

CALL statement

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

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

where *sub* is the name of the subroutine and *a* is an actual argument or argument list. Refer to Chapter 9, "Subprograms," for more information.

RETURN statement

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

```
RETURN [e]
```

where *e* is an optional integer expression that specifies an alternate statement in the calling program that is to receive control. Refer to Chapter 9, "Subprograms," for a complete description.

STOP statement

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

```
STOP [s]
```

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

Example:

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

PAUSE statement

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

```
PAUSE [s]
```

where

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

Example:

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

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

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

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

```
Execution resumed after PAUSE.
```

END statement

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

```
END
```

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

Input/output statements

7

Input/Output (I/O) statements provide a method for transferring data between internal storage and external media or between internal storage and internal files. CONVEX FORTRAN supports READ, *ACCEPT*, and *DECODE* statements for input, and WRITE, *TYPE*, PRINT, and *ENCODE* statements for output. Table 36 lists supported I/O statements by category.

Table 36
Data transfer I/O statements

Statement category	Statement name						
	READ	WRITE	*ACCEPT*	*TYPE*	PRINT	*DECODE*	*ENCODE*
Sequential/ External:							
Formatted	Yes	Yes	*Yes*	*Yes*	Yes	*No*	*No*
Unformatted	Yes	Yes	*No*	*No*	No	*No*	*No*
List-directed	Yes	Yes	*Yes*	*Yes*	Yes	*No*	*No*
Namelist-directed	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*No*
Sequential/ Internal:							
Formatted	Yes	Yes	*No*	*No*	No	*Yes*	*Yes*
List-directed	*Yes*	*Yes*	*No*	*No*	*No*	*No*	*No*
Direct/ External:							
Formatted	Yes	Yes	*No*	*No*	No	*No*	*No*
Unformatted	Yes	Yes	*No*	*No*	No	*No*	*No*

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

Note

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

Records

A sequence of characters or values processed as a unit constitutes a record; I/O statements transfer all data as records. Formatted records contain characters; unformatted records (those written without format specification) consist of bytes that represent binary values. Each unformatted I/O statement transfers one record. Formatted, list-directed, and namelist-directed I/O statements 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). 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. 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. A file is either internal (array or variable) or external (located on a peripheral device). There are two methods of accessing files: sequential and direct.

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

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

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`; 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.

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 `FD`nnn, 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.

Note

When the data format attribute (see the "Binary Datafile Format Conversions" section of this chapter) is specified, either in an `OPEN` statement or a shell variable, the source code must be compiled with CONVEX FORTRAN V6.0 or higher and linked with CONVEX FORTRAN V6.0 or higher libraries. If a version of CONVEX FORTRAN prior to V6.0 is used, unpredictable behavior, including program aborts and errors, can result.

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

Accessing files

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

Sequential access

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

Example:

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

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

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

Direct access

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

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

Example:

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

All records of a direct-access file have the same length. The record size is specified in bytes when the file is opened. Every time you read or write a record, you must specify a record number to indicate the record to be read or written.

I/O statement format

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

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

where

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

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

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

Input/Output lists

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

- Character substring name (`CHAR(6:10)`)
- Variable name (`L`)
- Array name (`MYEXAM`)
- Array element name (`M(3)`)
- Implied-DO list (`J,K,L,M, I=1,4`)
- An expression (`K + L` or `'JKL'`); used for output only. The expressions cannot contain function references with I/O statements in them.

When an array name without a subscript appears in an *iolist*, the elements are processed in the order in which they are stored, for example, `M(1,1)`, `M(2,1)`, and so on.

Implied-DO lists

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

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

where

dlist
: is an I/O list.

v
: is an integer or real variable.

ae_1, ae_2, ae_3
: are arithmetic expressions.

The variable and arithmetic expressions have the same forms and functions as those in the standard DO statements. The loop begins with the value of ae_1 and increments by the value of ae_3 until it equals or exceeds the value of ae_2. The loop then exits. Elements in *dlist* can reference *v*, but cannot change the value of *v*. The implied-DO loop can be nested.

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

Example:

```
      WRITE (7)(A,B,I=1,10)
C     WRITES THE PAIR A,B 10 TIMES

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

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

Specifiers

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

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

Unit specifier

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

```
[UNIT= ] u
```

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

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

Format specifier

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

```
[FMT =]f
```

or

```
[FMT =]*
```

where

f

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

*

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

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

Record specifier

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

A record specifier has the following forms:

```
REC = r
```

or

```
'r
```

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

Status specifier

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

```
IOSTAT = ios
```

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

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

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

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

Error specifier

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

```
ERR = s
```

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

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

End-of-file specifier

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

```
END = s
```

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

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

Namelist specifier

The namelist specifier specifies that namelist-directed I/O is to be used and specifies the group name for the entities that are modified during input or written on output. The namelist specifier has the following form:

```
NML = nlgrpname
```

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

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

READ statement

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

The statement has the following form:

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

where

clist

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

iolist

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

f

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

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

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

External sequential READ statements

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

Formatted

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

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

Example:

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

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

Unformatted

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

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

Example:

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

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

List-directed

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

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

Examples:

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

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

Namelist-directed

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

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

When the namelist-directed READ is used without specifying a control information list, it has the following form:

```
READ nlgrpname
```

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

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

Example:

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

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

External direct READ statements

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

Formatted

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

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

Example:

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

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

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

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

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

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

Example:

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

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

Internal READ statements

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

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

Sequential access

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

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

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

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

The following statement uses list-directed formatting.

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

Direct-access

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

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

Example:

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

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

ACCEPT statement

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

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

or

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

or

```
ACCEPT nlgrpname
```

where

f
: *is the non-keyword form of a format specifier.*

*
: *specifies list-directed formatting.*

iolist
: *is an I/O list.*

nlgrpname
: *is the non-keyword form of the namelist specifier.*

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

Example:

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

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

WRITE statement

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

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

where

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

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

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

The `WRITE` statement writes at least one record consisting of values from the I/O entities. The file is then positioned at the beginning of the next record. In the absence of an `OPEN` statement, the type of file written by CONVEX FORTRAN is dependent on the form of the `WRITE` statement used. If an `OPEN` statement is present and specifies `FORM='UNFORMATTED'` or `ACCESS='DIRECT'`, an unformatted file is written. Unformatted writes to internal files are not permitted. Direct-access, internal I/O is permitted. The logical record length is the length of the array element. A character variable array is similar to a fixed-length, direct-access file, and follows the same rules as formatted I/O.

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

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

In a FORTRAN 77 subroutine called from a C program, however, this statement writes to the file `fort.6` instead. This output appears on the terminal only if the environment variable `FOR006` is defined as `SYS$OUTPUT`.

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

Sequential-access `WRITE` statements

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

Formatted

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

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

Examples:

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

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

Unformatted

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

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

Examples:

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

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

List-directed

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

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

Example:

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

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

Namelist-directed

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

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

Example:

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

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

External direct-access WRITE statements

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

Formatted

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

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

Examples:

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

Both statements write variables D, E, and F to record number 25 of unit 50 according to the format specified in statement 100. The second statement also transfers control to statement 100 if an error condition exists.

Unformatted

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

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

Examples:

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

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

Internal WRITE statements

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

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

Sequential access

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

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

Examples:

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

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

Direct access

The internal direct-access WRITE statement has the form:

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

Example:

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

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

PRINT and TYPE statements

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

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

where

f
: is the format specifier.

*
: specifies list-directed formatting.

iolist
: is an I/O list.

nlgrpname
: *is the non-keyword form of the namelist specifier.*

Example:

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

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

Additional statements

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

ENCODE statement

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

The ENCODE *statement has the following form:*

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

where

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

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

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

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

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

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

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

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

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

Example:

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

Result:

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

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

DECODE statement

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

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

where

c

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

f

identifies the format (an error results if more than one record is specified).

b

is an array, array element, variable, or character substring reference that contains the characters to be translated to internal form.

ios

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

s

is the label of an executable statement.

iolist

is an I/O list that receives the data after translation to internal form.

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

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

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

Example:

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

Result:

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

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

FIND statement

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

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

where

u
: *is a logical unit number; it must refer to a direct-access file.*

r
: *is the direct-access record number; it cannot be less than 1 or greater than the number of records defined for the file.*

s
: *is the label of the executable statement to which control transfers if an error occurs.*

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

Example 1:

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

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

Example 2:

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

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

Auxiliary input/output statements

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

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

OPEN statement

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

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

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

Example:

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

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

Note

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

The following sections describe the OPEN statement keywords, which are summarized in Table 37. In the descriptions, the term "numeric expression" can be any integer or real expression. The value of the expression is converted to the INTEGER data type before it is used in the OPEN statement.

Table 37
OPEN statement keywords

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

Table 37
(continued)

Keyword	Values	Function	Default
RECORDTYPE or RT	*'FIXED'* *'VARIABLE'*	*Record structure*	*'VARIABLE' for sequential access; 'FIXED' for direct*
STATUS *or* *TYPE*	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
UNIT	E	Logical unit number	*N/A*

Key:
E is a numeric expression.

C is a character expression, *numeric array name, numeric variable name, or numeric array element name.*

V is an integer variable name.

S is a statement label.

ACCESS keyword

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

```
ACCESS = cex
```

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

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

Example:

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

ASSOCIATEVARIABLE keyword

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

```
ASSOCIATEVARIABLE = asv
```

where asv is an integer variable.

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

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

Note

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

BLANK keyword

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

```
BLANK = blnk
```

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

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

BLOCKSIZE keyword

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

```
BLOCKSIZE = bls
```

where bls is a numeric expression.

The default is the system default for the device. If you specify BLOCKSIZE, the physical record for block devices is set to the value of bls, with a maximum of 64 kbytes. For other devices (such as raw tape), the BLOCKSIZE value is rounded up to a multiple of the file system block size. An INQUIRE statement with a BLOCKSIZE keyword returns the rounded-up value.

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

Example:

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

CARRIAGECONTROL keyword

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

```
CARRIAGECONTROL = cc
```

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

The default is 'LIST' for formatted files and 'NONE' for unformatted files. 'LIST' transmits the first character of each formatted output record unchanged. FORTRAN replaces the first character of each formatted output record with the control characters required to interpret it on an ASCII output device. These control characters are **CTRL-L** for start of page, newline for double spacing, and null (no character) for single spacing.

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

DEFAULTFILE keyword

The `DEFAULTFILE` *keyword defines part of a default file specification. The keyword has the form:*

```
DEFAULTFILE = c
```

where c is a character expression.

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

Example 1:

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

This statement opens the file `info.dat.`

Example 2:

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

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

Example 3:

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

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

DISPOSE keyword

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

```
DISPOSE = dis
```

or

```
DISP = dis
```

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

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

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

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

ERR keyword

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

```
ERR = sl
```

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

FILE keyword

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

```
FILE = fln
```

or

```
NAME = fln
```

where *fln* represents a character expression. If the FILE specification does not appear in an OPEN statement, the unit is connected to a predefined file.

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

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

FORM keyword

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

```
FORM = f
```

where *f* represents the character expression with the value 'FORMATTED', 'UNFORMATTED', or 'PRINT'.

If you do not specify a format specifier in the OPEN statement, formatted I/O is assumed for sequentially accessed files. Unformatted I/O is assumed for direct-access files. *If FORM is specified as unformatted, then it can be followed by an optional data format qualifier:*

```
FORM = UNFORMATTED[/dataformat]
```

The dataformat qualifier specifies the format of the data file. Legal values are NATIVE (CONVEX-NATIVE or CONVEX_NATIVE), IEEE (CONVEX-IEEE or CONVEX_IEEE), VAX-D (or VAX_D), VAX-G (or VAX_G), CRAY, CRAYUB and USER-DEFINED (or USER_DEFINED). For more information on data file conversions, refer to the "Binary data file format conversions" section at the end of this chapter.

Data file conversion using the dataformat qualifier is most useful for programs that run only once or infrequently, programs that read or write small amounts of binary data, and programs whose data is irregular (the layout of the records within a file changes from one record to the next). If the amount of data that a program reads or writes is large and the layout of the data file is regular, consider writing a custom data conversion program instead.

Note

When the dataformat attribute is specified, either in an OPEN statement or a shell variable, the source code must be compiled with CONVEX FORTRAN V6.0 or higher and linked with the CONVEX FORTRAN V6.0 or higher libraries. If a version of CONVEX FORTRAN prior to V6.0 is used, unpredictable behavior, including program aborts and errors, can result.

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

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

IOSTAT keyword

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

```
IOSTAT = ios
```

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

MAXREC keyword

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

```
MAXREC = mr
```

where mr is a numeric expression.

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

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

Example:

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

NOSPANBLOCKS keyword

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

```
NOSPANBLOCKS
```

READONLY keyword

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

```
READONLY
```

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

RECL keyword

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

```
RECL = ie
```

or

```
RECORDIZE = ie
```

where *ie* is an integer expression with a positive value.

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

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

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

RECORDTYPE keyword

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

```
RECORDTYPE = typ
```

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

File access	*Default RECORDTYPE*
Direct	*FIXED*
Sequential	*VARIABLE*
List-directed	*VARIABLE*

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

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

Example:

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

STATUS keyword

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

```
STATUS = sta
```

or

```
TYPE = sta
```

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

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

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

UNIT keyword

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

```
[UNIT=] u
```

where *u* is a numeric expression. Valid logical unit numbers are 0 through 255.

If the unit number appears as the first parameter of the OPEN statement, the UNIT keyword can be omitted; otherwise, the UNIT keyword is required.

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

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

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

CLOSE statement

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

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

or

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

or

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

where

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

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

s
: is the label of an executable statement.

ios
: is an integer variable or integer array element.

The following statement

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

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

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

INQUIRE statement

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

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

or

```
INQUIRE ( [UNIT=]u, list)
```

where

fi

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

Note

The `FILE` name you specify when using the `INQUIRE` statement under the COVUEshell must be the absolute UNIX path name. VMS pathnames are not recognized.

list

is a list of specifiers that indicate the information to be determined for the file or unit. Each specifier appears in the list only once.
Table 38 describes the valid specifiers.

u

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

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

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

Example:

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

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

Example:

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

Table 38 enumerates and describes `INQUIRE` specifiers.

Table 38
INQUIRE specifiers

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

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

File-positioning statements

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

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

File-positioning statements have the following form:

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

or

```
REWIND u
```

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

or

```
BACKSPACE u
```

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

or

```
ENDFILE u
```

where

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

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

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

REWIND statement

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

Example:

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

BACKSPACE statement

The BACKSPACE statement positions the file connected to the specified unit before the preceding record. If the file is already at the first record, no action is taken. If the file is positioned after the endfile record, BACKSPACE positions the file before the endfile record. You cannot backspace a file that does not exist. Do not attempt to BACKSPACE in an unformatted sequential access Cray pure data file.

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

Example:

```
BACKSPACE 10
```

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

Example:

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

ENDFILE statement

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

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

Example:

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

Binary data file format conversions

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

CONVEX FORTRAN allows conversions of the data formats listed in Table 39.

Table 39
Data format conversion routine names

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

*See Appendix G for details on converting Cray files.

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

When to use the conversion feature

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

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

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

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

-dfc option

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

Conversion using OPEN statement

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

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

where *format* is one of the data format names listed in Table 39 and indicates the current data format of the file you want to convert. You can specify the data format in uppercase, lowercase, or mixed case.

When binary data is read from the file specified in the `OPEN` statement, the conversion routines convert the data from the specified data format to the data format implied by the program's mode, either `NATIVE` or `IEEE`.

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

Restrictions on conversions

Observe the following restrictions when using data file format conversions:

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

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

IEEE print a diagnostic message during execution and halt (unless an ERR= or IOSTAT= expression is specified in the I/O statement).

Note

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

- *The Cray does not support arithmetic items less than eight bytes long. So attempting to read or write these data types when the data format is CRAY or CRAYUB causes a diagnostic message to be issued at runtime (unless an ERR= or IOSTAT= clause is specified).*
- *When the FORM = UNFORMATTED/CRAY data format conversion is specified, only unformatted, direct access, unblocked ("pure") files can be read. You can convert sequential access, unformatted, blocked files into readable unblocked format with the fcUnblock utility. Refer to Appendix G, "Cray FORTRAN compatibility," for details on reading various unformated Cray files. Refer to the fcUnblock(3f) man page for more information on fcUnblock.*
- *Optimizing code containing type conversions (including implicit type conversions) can cause some code to vectorize poorly or not at all.*

Error handling using data format conversions

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

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

When an overflow or underflow occurs, then an appropriate value is stored in the user's variable for a READ or in the I/O buffer for a WRITE. The library then activates an overflow or underflow. If you ignore underflow and overflow, program execution continues. If not ignored, these conditions cause the program to receive a signal (SIGFPE) that terminates the program unless trapped via the signal(3) routine. You can call

`errtrap` to change the default handling of overflows and underflows. By default, programs ignore underflow (zero is substituted), and overflow causes a diagnostic message to be issued and terminates execution.

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

User-defined conversions

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

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

Table 40 shows a set of stub routines for user-defined data types. You only need to supply the routines that you actually use. If you forget to write a routine that is called, the stub routine prints an appropriate diagnostic message and halts execution of the program. This error message ignores ERR= and IOSTAT= clauses.

Each user-defined conversion routine is called as follows:

```
NAME ( source, target, bytecount)
```

where *source* and *target* are pointers to a block of storage, and *bytecount* is a pointer to an integer value. The routine is expected to pick up the values through the source pointer and store the converted data values through the *target pointer*. *bytecount* is the number of data values the routine converts multiplied by the size of one item in bytes. For example, a call of the form

```
CVT_INT4_TO_UD ( SOURCE , TARGET , 40 )
```

converts 10 INTEGER*4 values (40/4) from CONVEX binary integer format to the user-defined format. SOURCE points to the 10 values before conversion, and the user-supplied routine stores the values through TARGET.

The function must return 0 to indicate success and -1 to indicate failure. The function must also provide support for reserved values, infinity, overflows, and underflows as needed.

Sample conversion routine

The FORTRAN source code shown in Figure 23 converts `INTEGER*4` binary values into one's complement binary format. No provision is made in this routine for the maximum negative integer that cannot be represented in one's complement format.

Figure 23
FORTRAN example conversion routine

```
      INTEGER FUNCTION CVT_INTEGER4_TO_UD (SOURCE, TARGET, BYTECOUNT)
      INTEGER SOURCE(*), TARGET(*), BYTECOUNT
      INTEGER N
      N = BYTECOUNT / 4                      ! COMPUTE AN ITEM COUNT

      DO 10 I=1,N                            ! CONVERT N ITEMS
        TARGET(I) = SOURCE(I)                ! IF VALUE < 0, SUBTRACT 1
      IF (TARGET(I) .LT. 0) TARGET(I) = TARGET(I) - 1
10    CONTINUE

      CVT_INTEGER4_TO_UD = 0                 ! RETURN SUCCESS
      RETURN
      END
```

The C source code to accomplish the same task is shown in Figure 24.

Figure 24
C example conversion routine

```
int cvt_int_to_ud_ (source, target, byteCount)
   int * source, * target, * byteCount;
 {
   int n;
   n = ( *byteCount) / 4          /* Compute an item count            */
   while ( n-- > 0) {             /* Convert n items                  */
       *target = *source          /* Move the value                   */
       if (*target < 0)           /* If the value is negative,        */
          (*target)--;            /* Subtract one.                    */
       source++;                  /* Adjust the source & target pointers */
       target++;                  /* for the next value/destination   */
   }
   return (0);                    /* Return success.                  */
}
```

Note

A trailing underscore is required on the routine name if it is written in C.

Because some of the FORTRAN data types are not directly supported in C, it can be useful to declare the pointer's source *and* target *as something else, such as a* `char*` *or a* `struct` *pointer.*

Note

Only one user-defined data format is supported; therefore, a given program is restricted to one set of user-supplied conversion routines.

You can support several different user-defined data formats concurrently by modifying the program to save the current user-defined data format in a variable in COMMON*, which can be queried by each conversion routine.*

User-suppliedconversion routine names

Table 40 lists user-defined conversion routine names.

Table 40
User-supplied conversion routine names

FORTRAN Type	**Routine names**	
	FORTRAN-to-user-defined	**User-defined-to-FORTRAN**
INTEGER*1	cvt_integer1_to_ud	cvt_ud_to_integer1
INTEGER*2	cvt_integer2_to_ud	cvt_ud_to_integer2
INTEGER*4	cvt_integer4_to_ud	cvt_ud_to_integer4
INTEGER*8	cvt_integer8_to_ud	cvt_ud_to_integer8
REAL*4	cvt_real4_native_to_ud cvt_real4_ieee_to_ud	cvt_ud_to_real4_native cvt_ud_to_real4_ieee
REAL*8	cvt_real8_native_to_ud cvt_real8_ieee_to_ud	cvt_ud_to_real8_native cvt_ud_to_real8_ieee
REAL*16†	cvt_real16_native_to_ud cvt_real16_ieee_to_ud	cvt_ud_to_real16_native cvt_ud_to_real16_ieee
COMPLEX*8	cvt_complex8_native_to_ud cvt_complex8_ieee_to_ud	cvt_ud_to_complex8_native cvt_ud_to_complex8_ieee
COMPLEX*16	cvt_complex16_native_to_ud cvt_complex16_ieee_to_ud	cvt_ud_to_complex16_native cvt_ud_to_complex16_ieee
LOGICAL*1	cvt_logical1_to_ud	cvt_ud_to_logical1
LOGICAL*2	cvt_logical2_to_ud	cvt_ud_to_logical2
LOGICAL*4	cvt_logical4_to_ud	cvt_ud_to_logical4
LOGICAL*8	cvt_logical8_to_ud	cvt_ud_to_logical8
CHARACTER	cvt_character_to_ud	cvt_ud_to_character

† *REAL*16 is not supported in IEEE mode programs.*

For all floating point data types (REAL, COMPLEX, and so on), the program's mode (NATIVE or IEEE) has been included as part of the name. This allows the runtime library to trap an unintended conversion.

COMPLEX data types are really two adjacent REAL values. For COMPLEX*8, there are two adjacent REAL*4 values. Because a byte count is passed to the conversion routines, the conversion routines for COMPLEX*8 can usually be identical to the REAL*4 routines.

Conversion using a shell variable

Each time an OPEN statement is executed for a unit number not previously opened, the program's environment is searched for the shell variable named FORnnnOPEN, where nnn is the unit number. nnn must be 3 digits long with leading zeros as needed. If FORnnnOPEN is not found, the file attributes specified in the OPEN statement are used.

Any attributes not specified in the shell variable are taken from the OPEN statement. In the absence of an associated shell variable, the attributes specified with the OPEN statement are used.

All OPEN operations for that particular unit use those attributes specified in the shell variable.

Note

The data format attribute (dataformat=xxxx) is only valid if the file is opened for unformatted I/O. Otherwise, it is ignored.

Table 41 lists the attributes that you can specify in the shell variable.

Table 41
Shell variable attributes

Keywords & abbreviations	Legal Values
BLANK BLNK	null, zero
BLOCKSIZE BLKSZ	<number>
CARRIAGECONTROL	FORTRAN, LIST, NONE
MAXREC	<number>
RECL	<number>
RECORDTYPE RT	FIXED, VARIABLE
DISPOSE DISP	KEEP, DELETE
POSITION POS	as is, REWIND, APPEND
DATAFORMAT DF	NATIVE, IEEE, vax-d, vax-g, CRAY, CRAYUB, user-defined

Note

When the data format attribute is specified, either in an OPEN statement or a shell variable, the source code must be compiled with CONVEX FORTRAN V6.0 or higher and linked with the CONVEX FORTRAN V6.0 or higher libraries. If a version of CONVEX FORTRAN prior to V6.0 is used, unpredictable behavior, including program aborts and errors, can result.

Use the following `csh` *command to specify a* `FORnnnOPEN` *value, where* `nnn` *is a three-digit number from 000 to 255.*

```
% SETENV FORnnnOPEN "RECL=256, DATAFORMAT=VAX-G"
```

The corresponding `sh` *commands are as follows:*

```
$ FORnnnOPEN="RECL=256, DATAFORMAT=VAX-G"

$ EXPORT FORnnnOPEN
```

The corresponding COVUEshell command is:

```
$SET COVUE ENVIRON : FORnnnOPEN = "RECL=256, DATAFORMAT=VAX-G"
```

You can use keyword abbreviations to reduce the total length of the shell variable's value. Keywords and their abbreviations can be given in either uppercase or lowercase. The shell variable name must be in uppercase.

Format specifications

8

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

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

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

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

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

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

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

- An asterisk that indicates list-directed I/O, for example:

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

FORMAT statement

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

sl FORMAT (*flist*)

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

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

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

where

r
: represents the repeat count.

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

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

fs
: nonempty *flist*.

A repeatable edit descriptor has one of the following forms:

[*r*]*c*[*r*]*cw* [*r*]*cw.m* [*r*]*cw.d*[Ee]

[*r*]*cw.d*[De] [*r*]*cw.d.e*

where

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

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

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

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

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

E or D
: identifies the exponent field.

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

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

FORMAT control

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

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

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

Example:

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

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

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

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

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

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

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

Repeat count

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

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

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

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

Descriptors

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

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

A descriptor

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

A[*w*]

In an input statement, the A field descriptor transfers *w* characters from the external record and assigns them to the corresponding I/O list element. If the *w* field is not specified, the size equals the length of the character variable, character

substring reference, or character array element. For numeric I/O list elements, the size depends on the data type, as shown in Table 42.

Table 42
Character assignment for numeric I/O list elements

I/O List elements	Maximum no. of characters
*LOGICAL*1*	*1*
*LOGICAL*2*	*2*
LOGICAL*4	4
*LOGICAL*8*	*8*
*INTEGER*1*	*1*
*INTEGER*2*	*2*
INTEGER*4	4
*INTEGER*8*	*8*
REAL*4	4
REAL*8(DOUBLE PRECISION)	8
*REAL*16*	*16*
COMPLEX	8
COMPLEX*16(DOUBLE COMPLEX)	*16*

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

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

The following example illustrates reading into a CHARACTER*5 variable.

Example:

Format code	External field	Internal value
A	COMVEXCOMPUTER	CONVE
A4	CONVEXCOMPUTER	CONV^
A14	CONVEXCOMPUTER	PUTER

The following example illustrates writing from a CHARACTER*10 variable.

Example:

Format code	Internal value	External field
A	MY^EXAMPLE	MY^EXAMPLE
A4	MY^EXAMPLE	MY^E
A14	MY^EXAMPLE	^^^^MY^EXAMPLE

Apostrophe descriptor

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

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

produce

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

Asterisk descriptor

CONVEX FORTRAN supports the asterisk descriptor under the -cfc option. The asterisk descriptor is used to delimit literal text and behaves identically to the apostrophe descriptor. Use two consecutive asterisks to produce a single asterisk.

H descriptor

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

This descriptor has the following form:

```
nHc...c
```

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

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

produce

```
ENTER 'FILE' NAME
```

L descriptor

The L descriptor formats logical variables and has the form:

```
Lw
```

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

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

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

Input examples:

Format code	External field	Internal value
L2	T60	.TRUE.
L7	^^FALSE	.FALSE.
L7	1234567	Error-invalid

Output examples:

Format code	Internal field	External field
L1	.TRUE.	T
L3	.FALSE.	^^F

I descriptor

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

I*w*

or

I*w.m*

where

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

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

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

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

Input examples:

Format code	External field	Internal value
`I3`	`760`	`760`
`I4`	`^^^^`	`0`
`I4`	`-760`	`-760`
`I5`	`760^^`	`760`
`I5`	`760^^`	`76000` (blanks interpreted as zeros)
`I5`	`7.60^^`	Error: decimal

Output examples:

Format code	Internal value	External field
`I4`	`760`	`^760`
`I8.4`	`760`	`^^^^0760`
`I3`	`-760`	`***`
`I4`	`0`	`^^^0`
`I4.0`	`0`	`^^^^`

o descriptor

The o descriptor transfers unsigned octal values. The descriptor has the form:

ow [.m]

where

w

is an unsigned, positive integer constant that specifies the field to be edited is w characters in width.

m

is an unsigned, integer constant that specifies the minimum number of digits for output, including leading zeros if necessary.

On input, format code o *transfers w characters from the external field and assigns them as an octal value to the corresponding I/O list element. On output, if m is specified and the external field contains fewer digits than m, the remaining positions are padded with zeros on the left. You can only use the numerals 0 through 7 in the external field; you cannot use a decimal point (.), a sign (+ or -), or an exponent field.*

Input examples:

Format code	External field	Internal octal
O3	523	523
O4	23176	2317
O4	2.317	Error: decimal
O4	-1234	Error: signed

Output examples:

Format code	Internal decimal value	External value
O6	4095	^^7777
O6	-4095	*****
O3	4095	***
O4.3	8	^010

z descriptor

The z *descriptor transfers unsigned hexadecimal values and is represented as*

Zw [.m]

where

w

is an unsigned, positive integer constant that specifies the field to be edited is w characters in width.

m

is an unsigned, integer constant that specifies the minimum number of digits for output, including leading zeros if necessary.

On input, descriptor `Z` *transfers w characters from the external field and assigns them as a hexadecimal value to the corresponding I/O list element. On output, if m is specified and the value has fewer digits than m, the remaining positions are padded with zeros on the left. You can use only the numerals 0 through 9 and the letters* `A` *(*`a`*) through* `F` *(*`f`*) in the external field; you cannot use a decimal point (.), a sign (+ or -), or an exponent field.*

Input examples:

Format code	External field	Internal hex value
Z3	9A1	9A1
Z3	9A1B	9A1
Z3	9A.1	Error: decimal

Output examples:

Format code	Internal decimal value	External value
Z4	4095	^fff
Z5	-1	*****
Z6.4	4095	^^0fff
Z2	4096	**

`F` descriptor

The `F` descriptor provides formatting of real numbers. It has the following form:

```
Fw.d
```

where

w
: is an unsigned, positive integer constant that specifies the field to be edited is *w* characters in width.

d
: specifies the number of digits in the fractional (right of the decimal) part of the real number.

During input, the processor transfers *w* characters from the external field and stores the real values in the corresponding I/O list elements. The input field consists of an optional sign followed by a string of digits that can contain a decimal point. If the field has a decimal point, the *d* term has no effect; the location of the explicit decimal overrides the location specified by the field descriptor. If you omit the decimal point and the exponent, the rightmost *d* digits are interpreted as the fractional part of the field with leading zeros assumed if necessary.

On input, leading blanks are not significant; nonleading blanks are interpreted according to the `BLANK` specifier or the `BZ` or `BN` descriptor. If the field contains only blanks, the value is zero. The processor treats a plus sign (+) or no sign as a positive value; a minus sign (-) indicates a negative value.

During output, the processor transfers the value of the I/O list element rounded to *d* decimal positions and outputs it in a field *w* characters wide, right justified. *w* must include a space for a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal, for example, at least equal to or greater than $d + 3$. Leading spaces are added, if needed, to fill the field.

Input examples:

Format code	External field	Internal value
F8.5	1234567^	12.34567
F8.5	12345.67	12345.67
F8.0	-1.23E-3	-.00123
F8.5	123456789	123.45678

Output examples:

Format code	Internal value	External field
F9.4	123.456789	^123.4568
F5.2	123.456789	*****
F6.3	+1.12	^1.120
F6.3	-1.12	-1.120

If the value is too large for the field, asterisks (*) are output. In native format, if the sign is 1 and the exponent is 0, `Rop` (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, `NaN` (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, `Inf` (infinity) is output.

E and D descriptors

The `E` and `D` descriptors are functionally identical. Both transfer real values in exponential form and edit external `REAL`, `DOUBLE PRECISION`, or complex data. These descriptors differ only in the exponent symbol they use. The `E` and `D` descriptors have the following forms:

`E`*w.d*`,` `E`*w.d.e*`,` or `E`*w.d*`E`*e*
`D`*w.d*`,` `D`*w.d.e*`,` or `D`*w.d*`E`*e*

where

w
: is the width of the field containing the real number. The width is the count of all characters in the field, including sign (if any), decimal point, and exponent.

d
: is the fractional part of the field that contains *d* digits.

E or D

: identifies the exponent part that contains *e* digits (no effect on input).

e

: indicates the number of digits in the exponent.

On input, the descriptors read *w* characters from the external field and assign them as a real value to the corresponding I/O list element. The values being read consist of a string of digits with an optional decimal point. When the decimal point is included, the *d* term has no effect. When the decimal is omitted, however, the least significant *d* digits of the string, not including the exponent, are considered the fractional part of the value.

On output, the E and D descriptors transfer the value of the I/O list element rounded to *d* decimal positions and output it to a field *w* characters wide, right justified. *w* must include space for a minus sign (when necessary), the decimal point, *d* digits to the right of the decimal, a two- or three-digit exponent (depending on whether the I/O list item is REAL*4 or REAL*8), E or D, and the sign of the exponent.

All data values, including *REAL*16*, can be used with the E and F formats.

In a data value, the exponent can be a signed-integer constant, or E or D followed by zero or more blanks, followed by an optional signed-integer constant. You can use the legal characters—digits 0 through 9, decimal point, plus, minus, E, D, and blank. With the descriptors in the form of E*w.d*, E*w.d.e* or E*w.d*E*e* (D*w.d*, D*w.d.e*, D*w.d*E*e*), the value of the next item in the list has the following form:

[±] [0] .x_1, x_2 ... x_d *exp*

where

±

: signifies a plus or minus sign; the plus sign is optional for a positive value.

0

: an optional leading zero.

$x_1, x_2 \ldots x_d$

: are the *d* most significant digits of the value after rounding.

exp

is a decimal exponent that is of the form $E \pm z_1 [z_2] \ldots z_{e3}$ where z is a digit and there are *e* digits in the exponent.

Input examples:

Format code	External field	Internal value
`E4.3`	`.625`	`.625`
`E6.1`	`.78-01`	`.078`
`D6.3`	`-.62D4`	`-6200`
`D6.3`	`123456`	`123.456`
`E4.3`	`62567`	`6.256`

Output examples:

Format code	Internal value	External field
`D11.4`	`-6250.`	`-0.6250D+04`
`E10.3`	`625`	`^0.625E+03`
`E10.4`	`0.4568`	`0.4568E+00`
`E10.3`	`0.4568`	`^0.457E+00`
`E5.3`	`24.53`	`*****`
`E7.2.1`	`2.718`	`0.27E+1`

If the value is too large for the field, asterisks (`*`) are output. The default length for the exponent field for `REAL*16` numbers is three. In native format, if the sign is 1 and the exponent is 0, `Rop` (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, `NaN` (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, `Inf` (infinity) is output.

G descriptor

The `G` descriptor edits external single-precision, double-precision, quad-precision, or complex data. It has the following form:

```
Gw.d, Gw.d.e or Gw.dEe
```

where

w

is a nonzero, unsigned, integer constant that indicates the field width in characters.

d

is a nonzero, unsigned, integer constant that indicates the number of significant digits to be printed. On output, when the range of the value to be printed forces `E` style editing (see below), *d* specifies the number of characters to the right of the decimal point. When the range of an output value allows `F` editing, *d* specifies the number of significant digits to print.

`E`

identifies the exponent field.

e

is an unsigned, integer constant that indicates the number of digits in the exponent.

Input editing is identical to `F`, `E`, and `D` editing. You can use the `G` descriptor when you are not certain that the values you are using can be adequately represented by the `F` descriptor because of their magnitude—either too large or too small.

On output, the `G` descriptor uses either the `F` or `E` style of editing depending on the magnitude of the value. If the value can be represented using the `F` format without loss of significant digits, `F` is chosen; otherwise, `E` is chosen.

Assume `M` is the magnitude of the data in the field. If `M` is less than 0.1 or greater than or equal to `10**d`, the output editing of `G`*w.d* or `G`*w.d*`E`*e* is the same as that of *k*`PE`*w.d* and *k*`PE`*w.d*`E`*e*, respectively, and *k* is the scale factor currently in effect. If `M` is greater than or equal to 0.1 or less than `10**d`, however, the `F` mode of editing is used with output of the four-character exponent field as four blanks after the value. The scale factor has no effect and the value of `M` determines the editing as follows shown in Table 43.

Table 43
Data conversion based on magnitude

Magnitude of data	Conversion equivalence
`0.1 ≤ M < 1.0`	`F(`*w-n*`).`*d*`,`*n*`(' ')`
`1.0 ≤ M or < 10.0`	`F(`*w-n*`).(`*d-1*`),`*n*`(' ')`
.	.
.	.
.	.
`10**(d-2) ≤ M < 10**(`*d*`-1)`	`F(`*w-n*`).1,`*n*`(' ')`
`10**(d-1) ≤ M < 10**`*d*	`F(`*w-n*`).0,`*n*`(' ')`

The value *n*(' ') specifies that four or *e* + 2 spaces are to follow the numeric data representation; *n* is 4 for G*w*.*d* and *e* + 2 for G*w*.*d*E*e*. Be sure the *w* term is large enough to include a sign, if necessary, a decimal point, *d* digits to the right of the decimal and either a 4-character or an (*e* + 2)-character exponent. Thus, you must make *w* equal to or greater than *d* + 7 or *d* + 5 + *e*.

Input examples:

Format code	External field	Internal value
G8.5	^1234567	12.34567
G8.5	12345.67	12345.67
G8.0	-1.234-3	.001234

Output example:

Format code	Internal value	External field
G13.6	-1234	^-1234.00^^^^
G13.6	0.01234	^0.123400E-01
G13.6	1.23456789	^^1.23457^^^^
G10.4	15.65	^15.65^^^^
E10.4	15.65	0.1565E+02
F10.4	15.65	^^^15.6500

If the value is too large for the field, asterisks (`*`) are output. The default length for the exponent field for *REAL*16* numbers is three. In native format, if the sign is 1 and the exponent is 0, `Rop` (reserved operand) is output followed by the fraction in hexadecimal. In IEEE format, if the exponent is all ones and the fraction is nonzero, `NaN` (not a number) is output followed by the fraction in hexadecimal; if the exponent is all ones and the fraction is 0, `Inf` (infinity) is output.

B descriptors

The B descriptors operate only during execution of the input statements and affect only the numeric descriptors I, O, Z, F, E, D, and G. The BN and BZ descriptors supersede the default interpretation of blanks while the *B descriptor causes return to the default mode of blank interpretation.* Their forms and meanings are:

B

reverts to default interpretation.

BZ
interprets blanks as zeros.

BN
interprets blanks as nulls.

When execution of a formatted input statement begins, blanks can either be interpreted as zeros or ignored, depending on the value of the BLANK specifier in the OPEN statement. The BLANK specifier can be omitted from the OPEN statement, and the OPEN statement itself can be omitted. Additionally, the -vfc compiler option and use of the COVUEshell can effect the default value of BLANK. Table 44 shows the value of the BLANK specifier under various definition states.

Table 44
BLANK specifier defaults

OPEN/BLANK definition		BLANK specifier value	
OPEN stmt	BLANK spec.	Under COVUEshell or using the -vfc flag	-vfc and COVUEshell not used
Omitted		BLANK = 'ZERO'	BLANK = 'NULL'
Included	Omitted	BLANK = 'NULL'	BLANK = 'NULL'

The BN descriptor causes the processor to treat all embedded blank characters as nulls in subsequent input fields for the current statement. When the processor encounters the BN specifier, it treats the input field as though the embedded blanks have been moved to the position of leading blanks; the remainder of the field becomes right justified. The processor assigns the value of zero to a field containing only blanks. If you specify the BZ descriptor, the processor treats all embedded and trailing blanks in subsequent numeric input fields as zeros.

For example, if a file connected to unit 5 contains the record

```
^^^5^-500^^^3^056
```

and the BLANK specifier has a NULL value or the BN descriptor is specified, the statements:

```
READ (5,'(I4,I7,I6)') L, M, N
READ (5,'(BN,I5,I7,I6)') L, M, N
```

assign the value of 5 to L, the value of -500 to M, and the value of 3056 to N. The processor ignores all blanks. If the BZ descriptor is indicated by:

```
READ (5,'(BZ,I5,I7,I6)') L, M, N
```

the values assigned become L = 50;M = -50000;N = 30056. The processor treats all nonleading blanks as zeros. If another input statement refers to unit 5, blank interpretation returns to the default value.

The descriptor B causes a return to the default mode of blank interpretation ('NULL') and is identical to BN. For example, change the previous example to include a B descriptor:

```
READ (5,'(BZ,I5,I7,B,I6)') L, M, N
```

The value of 3056 is assigned to N rather than 30056, because the B descriptor returns blank interpretation to default mode.

P descriptor

The P descriptor specifies a scale factor for real and complex values. The P descriptor can be used on input or output and applies to the F, E, D, and G edit descriptors. The P descriptor has the form:

```
nP
```

where *n* is an optional integer constant that specifies the number of positions, to the left or right, that the decimal point is to be moved. The value can be signed. Its default value is 0.

If no P descriptor is specified, a scale factor of 0 is assumed. Once specified, a scale factor remains in effect within a FORMAT statement until another P descriptor is encountered. The following example uses a scale factor of 0 for the first format descriptor and a scale factor of 2 for the two remaining descriptors.

Example:

```
      PRINT 50, D
50    FORMAT (F8.2,2PF8.2,F6.2)
```

On input, the scale factor (with the F, E, D, or G descriptors) causes the externally represented number to be multiplied by 10**-*n* before it is assigned to the I/O list element.

Examples:

Format code	External field	Internal value
3PF7.4	56.789^	.056789
-3PE6.3	56.789	56789.

On output, when you use the scale factor with the `F` descriptor, the externally represented number equals the internally represented number multiplied by `10**`*n*. When the scale factor is used with `E` or `D`, the nonexponent part of the constant is multiplied by `10**`*n* and *n* is subtracted from the exponent. With `G`, if the `F` style of formatting is used, the scale factor is ignored; otherwise, the effect is the same as `E` editing.

Examples:

Format code	Internal value	External field
-1PF7.3	58.967	^^5.897
2PE10.3	890.11	^89.01E+01

If you use a scale factor when an external field has an explicit exponent, for example, `5.E02`, the processor ignores it; in this case, `500` is stored regardless of the scale factor.

S descriptors

The `S` descriptor can be used to control optional plus (+) characters in numeric output *or to cause integer values to be interpreted as unsigned during output conversion.* If you do not use an `S` descriptor, positive values do not have leading plus signs. The `S`, `SP`, and `SS` descriptors act only during statement execution and only with `I`, `F`, `E`, and `D` editing descriptors. *The `SU` descriptor only affects integer values.* The descriptors have the following forms:

`S`
: reverts to normal interpretation.

`SP`
: adds a plus sign (+).

`SS`
: suppresses plus signs.

`SU`
: *outputs integer values as unsigned values.*

The SP descriptor forces a plus sign during output for all subsequent positive I, F, D, E, and G values within the format specification. Include space for the plus sign in the numeric fields. When you use the SS descriptor, the processor suppresses leading plus sign characters from any position where the plus sign is optional. The S descriptor returns the normal plus sign handling option to the processor. For example, if L = +5, M = 100, N = -10, I = 50, J = 6000, and K = -450, the statements:

```
      WRITE (10,30) L, M, N, I, J, K
30    FORMAT (SS,I2,I5,SP,I4,I4,S,I5,I5)
```

write the record as:

```
^5^^100^-10^+50^6000^-450
```

The SU descriptor causes integer values to be interpreted as unsigned during output conversion. SU remains in effect until another sign-control specifier is encountered or until format interpretation is complete. It has no effect on input. Radix and unsigned specifiers can be used to format a hexadecimal dump as follows:

```
2000 FORMAT (SU, 16R, 8I10.8)
```

R descriptor

The R descriptor changes the radix for integer I/O. This descriptor applies only to integers (I descriptor) and must not be used with other descriptors. The R specifier has the form

```
[n]R
```

where $2 \leq n \leq 36$. The default value is 10. Omitting n restores the default decimal radix. The radix specified by R remains in effect until another radix is specified or until format interpretation is complete.

Example:

```
      I = 15
      WRITE (6,10) I, I, I
10    FORMAT (16R,I4,8R,I4,R,I4)
```

produces

```
^^^F^^17^^15.
```

X descriptor

The X descriptor sets the position in a record and has the form:

*n*X

where *n* indicates the number of character positions to move forward (skip over) from the current position in the file. The value of *n* must be greater than or equal to 1. The default is 1.

The X descriptor is functionally identical to the TR descriptor. When you use the X descriptor, *n* indicates that the next *n* characters are to be skipped. The character following the number of skipped positions is transmitted. For example, the statements

```
      WRITE (*,200) 450, 8921
  200 FORMAT (2X,I3,3X,I4)
```

insert two blanks before 450 and three blanks before 8921.

The X format descriptor cannot in itself change the length of a record. For example, the statements

```
      WRITE (6,100) I
  100 FORMAT (I4,X)
```

produce a four-character record followed by a newline, not a trailing blank.

T descriptors

The T (tab) descriptors control forward and backward positioning within a record for input or output of characters. These descriptors let you skip portions of a record or reread portions of a record. The T descriptors are T, TR, and TL.

The T descriptor has two forms; the first form is

T*n*

where *n* specifies the absolute position within the record. This form indicates transmission of characters at position *n*. For example, if a file connected to the designated input unit contains the record:

```
^2.5^200^^40
```

then execution of the statements

```
      READ (*,35) A, B
35    FORMAT (T2,F3.0,T11,F3.0)
```

assigns A the value of 2.5 (positions the file at character 2 and reads the next 3 characters according to format specification F3.0) then assigns the value of 40 to B (positions the file at character 11 of the record and reads the next 3 characters).

On output, for example, the following statements

```
      PRINT 50
50    FORMAT (T10,'MY',T13,'EXAMPLE')
```

output MY at position 10 and EXAMPLE at position 13.

Another form of the T descriptor is

```
T or nT
```

which causes tabbing to the next (or nth) 8-column tab stop. You can, therefore, align columns of alphanumerics without counting. For example, the following statements:

```
      READ (5,50) K,N
50    FORMAT (T,I4,2T,I3)
```

cause K to be read starting in character position 8 of the current record; the value for N is read starting in position 24 of the current record.

The second of the T-series descriptors has the following form:

```
TLn
```

where *n*, an unsigned, integer constant, indicates that the record is repositioned *n* characters left (backwards) from the current position in the record. The default is 0. For example, if the external record is 1.2345, the statements:

```
      READ (5,20) X, I
20    FORMAT (F6.0,TL4,I3)
```

produce X = 1.2345 and I = 234.

The third T-series descriptor has the following form:

```
TRn
```

where *n* is an unsigned integer that specifies the number of characters to move right (forward) from the current position in the record. The default is 0. The TR and X field descriptors are identical. For example, assume the external record is

```
12.345^^^^123
```

The statements

```
      READ (5,20) X, I
20    FORMAT (F6.0,TR4,I3)
```

produce X = 12.345 and I = 123.

The T descriptor cannot itself change the record length. For example, the statements

```
      WRITE (6,10) I
10    FORMAT (I4,TR10)
```

produce a 4-character record with no trailing blanks.

$ descriptor

The $ descriptor suppresses the newline at the end of the current record of a formatted sequential write. (In an input statement, the $ descriptor is ignored.) For terminal I/O, a typed response follows the output on the same line. For example, the following statements

```
WRITE (*,'(" enter value for x: ",$)')
READ (*,*) x
```

write `^enter value for x:^` *to the output device with the cursor positioned one space to the right of the colon.*

Q descriptor

The Q descriptor determines the number of unread characters in the current record. It is represented by:

```
Q
```

Example:

```
      READ (4,100) J, MYEXAM, (ISAM(I), I = 1, MYEXAM)
  100 FORMAT (I5,Q,80AI)
```

This example reads the first field into variable `J`, stores the number of remaining characters in `MYEXAM`, and causes transfer of that number of characters to the character array, `ISAM`. If you place `Q` first in the format specification, you can determine the actual length of the record.

In an output statement, the descriptor `Q` causes the corresponding I/O list element to be skipped.

Colon descriptor

The colon (`:`) descriptor ends format control when no items remain in the I/O list. If items remain in the I/O list, the colon descriptor has no effect. The following example:

```
    M = 15
    WRITE (10,40)M
 40 FORMAT (I2,:,' SAMPLE',I3)
```

writes `15` only, ending format control at the colon. Change the statements slightly, however, and the colon descriptor has no effect.

Example:

```
    M = 15
    N = 500
    WRITE (10,40) M, N
 40 FORMAT (I2,:,' SAMPLE',I4)
```

writes `15 SAMPLE 500`; the colon descriptor is ignored because items remain in the I/O list.

Slash descriptor

The slash (/) descriptor indicates the end of data transfer for the current record. For example, the following statements:

```
    READ (10,50) L, M, N
 50 FORMAT (I2/I4,I3)
```

cause L to be read from the first record, and M and N from the second record.

During input, use sequential slashes to indicate bypassing of records. The first slash indicates the end of input for the current record; subsequent slashes skip records. When you use the slash on a unit connected for sequential access, the remainder of the current record is skipped and the file is positioned at the beginning of next record. On direct access, 1 is added to the record number and the processor reads that record.

On output, slashes can be used to create empty records. The first slash indicates end of output for the current record; subsequent slashes produce empty records.

Default field descriptor values

If you do not specify a field width value with the field descriptors I, O, Z, L, F, E, D, G, or A, default values for *w*, *d*, and *e* are supplied based on the data type of the I/O list element. These default values are shown in Table 45.

Table 45
Default field descriptors

Field descriptor	List element type	*w*	*d*	*e*
I, O, Z	INTEGER*1, LOGICAL*1	7		
I, O, Z	INTEGER*2, LOGICAL*2	7		
I, O, Z	INTEGER*4, LOGICAL*4	12		
I, O, Z	INTEGER*8, LOGICAL*8	23		
O, Z	REAL*4	12		
O, Z	REAL*8	23		
O, Z	REAL*16	44		
L	LOGICAL	2		
F, E, G, D	REAL, COMPLEX*8	15	7	2
F, E, G, D	REAL*8, COMPLEX*16	24	15	3
F, E, G, D	REAL*16	42	33	3
A	LOGICAL*1, INTEGER*1	1		
A	LOGICAL*2, INTEGER*2	2		
A	LOGICAL*4, INTEGER*4	4		
A	LOGICAL*8, INTEGER*8	8		
A	REAL*4, COMPLEX*8	4		
A	REAL*8, COMPLEX*16	8		
A	REAL*16	16		
A	CHARACTER*n	n		

Comma field separator on input data

A comma between numeric fields overrides the width specified in the field descriptor. Because you can use a comma to end a field, you can avoid padding the input field, which makes entering data from a terminal keyboard easier. A comma field separator can be used with the numeric descriptors (`I`, `O`, `Z`, `F`, `E`, `D`, `G`, and `L`).

Example:

```
      READ (5,100) I,K
  100 FORMAT (2I4)
```

Record:

```
  1,2
```

Result:

```
  I = 1
  K = 2
```

The following constraints apply.

- *Two successive commas constitute a null field.*
- *You cannot use a comma to end a field that is controlled by an `A` or a character constant field descriptor. If the record reaches its physical end before w characters are read, short-field termination occurs and the characters you input are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list item.*

Runtime formats

Format specifications, called runtime formats, can be stored in character variables, character substrings, and character expressions, and in character arrays, *numeric arrays and numeric array elements. Numeric arrays and numeric array elements are treated as Hollerith constants.*

You can define or modify a runtime format during program execution. A runtime format is similar to a `FORMAT` statement but does not have a label or the word `FORMAT`.

Example:

```
INTEGER*8 IFMT
CHARACTER*8 SFMT
IFMT = 8H(2X,I12)
SFMT = '(2X,I12)'
WRITE (6,IFMT) I
WRITE (6,SFMT) I
```

Cray-style asterisk descriptors can be used in runtime formats in all modes; the `-cfc` option is not required when asterisks appear in runtime formats.

Variable formats

A variable format has an expression, enclosed in angle brackets, that is computed each time it is encountered during format scanning. The expression has the form:

<expression>

The *expression* in the angle brackets can be used in a FORMAT statement wherever you can use an integer, except as the character count of a Hollerith (H) descriptor.

A variable expression in a format statement is subject to the following rules:

- If the expression is not an integer, it is converted to an integer before use.
- Any valid FORTRAN expression can be used, including function calls and dummy argument references.
- The value of the expression must conform to the restrictions on magnitude that apply to its use in a format.
- A variable expression is not allowed in a runtime format.

Do not perform I/O operations within a function call used in a variable format expression or a runtime error occurs.

Example:

```
C TEST OF D AND E DESCRIPTORS WITH REPEAT COUNT
    1 FORMAT(<J+2>D10.4,<J/2+1>E10.4)
    2 FORMAT(<J+2>D10.4.2,<J/2+1>E10.4.2)
      J = 2
      READ(5,1) A,B,C,D,E,F
      WRITE(6,2) A,B,C,D,E,F
      STOP
      END
```

List-directed formatting

List-directed formatting transfers data based on the data type of the entity. A list-directed I/O statement uses an asterisk (*) as the format indicator. For example, the following statement:

```
READ (5,*) J, M, L
```

reads three fields from unit 5 and assigns integer values to the variables `J`, `M`, and `L`.

The list-directed record is a sequence of values and value separators. A value is generally a constant but can also be a null value, or the value can have the form

*r*c* or *r**

where

r

is an unsigned, nonzero, integer constant that represents the repeat count.

*r*c*

represents successive appearances of the constant *c*. You can enter a repeat count to assign a value to more than one entity with *r*c*.

*r**

repeat count with an empty constant (null value). A null value indicates that the value of the corresponding I/O entity is to remain unchanged.

Separators divide the values in each list-directed record. A value separator is a blank, comma, or a slash optionally enclosed by blanks. Normally, the blanks are considered part of some value separator. In the following cases, the blanks are not considered part of a value separator:

- Leading blanks in the first record, unless followed by a slash or comma.
- Blanks embedded in a character constant.

Input

You can use list-directed input from any file that allows formatted input. The data type of the constant, which can be logical, integer, real, complex, or character, determines the data type of the value, as well as the translation from external to internal form. A character list element must correspond to a character constant; likewise, a numeric element must correspond to a numeric constant. If the data type of the external numeric field does not match the data type of the numeric list item, the external value is converted according to the rules for conversion on assignment (see Chapter 5). Input fields are separated by blanks, commas, or slashes.

The format of a complex value is: left parenthesis followed by a numeric value, a comma, another numeric value (an ordered pair of numeric fields separated by a comma), followed by a right parenthesis. The processor ignores one or more blanks around either parenthesis or the comma. The end of record can occur between the real part and the comma or between the comma and the imaginary part.

Character input

Character constants for list-directed input are usually enclosed in apostrophes. Character constants can span record boundaries.

Embedded blanks, commas, and slashes within a character string are not considered separators. To include an apostrophe as part of a character string, use two consecutive apostrophes without an intervening blank or end of record.

The processor transfers the leftmost characters read, either truncating the constant to fit in the list item or filling it on the right with blanks.

CONVEX FORTRAN allows input of a string not enclosed in quotes. The string must not start with a digit and cannot contain a separator consisting of a right or left parenthesis, or blank

(space or tab). A newline ends the string unless escaped with a slash (\). Any string not meeting these restrictions must be enclosed in single or double quotes.

Nulls and slashes

You can specify a null value for a list item with a comma or with *r** in the external record. No characters between successive value separators or no characters preceding the first value separator indicate a null field. When assigning a null for the first value, you can use one comma; for a subsequent null, use two consecutive commas.

A null value does not alter the value of the corresponding input list item.

When the processor encounters a slash on list-directed input, it skips the rest of the I/O list items and ends the READ statement. Those items skipped retain their original values.

Namelist-directed formatting

To assign input values for a namelist-directed READ, you must delimit the input record (or records) with a dollar sign ($). Namelist input has the following form:

```
$nlgrpname [ent = value [,] ] ... $[END]
```

where

$
: *indicates the beginning and end of input. You can use the ampersand (&) rather than the $.*

nlgrpname
: *is the name defined for the entities contained in the namelist.*

ent
: *is a namelist entity. The entity can be a variable, an array name, a subscripted variable, a variable with a substring, or a subscripted variable with a substring.*

value
: *is a constant, a list of constants, or a repetition of constants or null values.*

END
: *is an optional delimiter indicating no more input.*

Use constant values for assigned values, array subscripts, and substring specifiers; you cannot use `PARAMETER` *constants.*

You can use any data type. Conversion (following rules of arithmetic assignment) is performed if the data type of a namelist entity and its assigned constant value do not match. Conversion between numeric and character data is not allowed.

For the `NAMELIST` *statement*

```
NAMELIST/SAM/ NAME, EXAM1, EXAM2, EXAM3
```

the following example shows how to input data to the namelist entities. You can assign the values in any order.

```
$SAM NAME='TESTA', EXAM1=5.2, EXAM2=6.78,
EXAM3=10.0 $
```

Several acceptable formats for entering input exist. For example, you can also enter the previous input as

```
$SAM^NAME='TESTA'^EXAM1=5.2^EXAM2=6.78^EXAM3=10.0
$END
```

You can use commas, tabs, and spaces as valid separators in the list of value assignments, and input can begin in any column. You cannot use nonblank control characters in column 1.

The previous example assigns values to all of the namelist entities associated with `SAM`*; however, you do not need to assign values to all the defined entities. Only those entities that you assign a value to change; those defined in the namelist but not assigned a value in the input data remain unchanged. Likewise, when you have defined character substrings and array elements in the namelist, only those you specify to receive input are changed. You can change part of a character substring. For example, to change the character variable* `NAME` *from* `'TESTA'` *to* `'TESTB'`*, use the following namelist-directed input:*

```
$SAM NAME(5:) = 'B' $END
```

The value for `NAME` *is '*`TESTB`*'; the first four positions of the value remain unchanged.*

When you assign values to an array name, the first value is associated with the first element, the second value with the second element, and so on. The number of array elements you can assign must be less than or equal to the size of the array.

Assume a program with the following statements:

```
DIMENSION MYRAY(10)
NAMELIST /SAM2/ MYRAY
READ SAM2
```

Assume that the input is

```
$SAM2 MYRAY = 10, 8, , 70 $END
```

On execution, the READ statement assigns the following values to the array elements:

MYRAY(1)	10
MYRAY(2)	8
MYRAY(3)	Unchanged
MYRAY(4)	70
MYRAY(5-10)	Unchanged

Values MYRAY(3) and (5-10) remained unchanged because the two consecutive commas in the input indicate not to change the current value, and values for unspecified array elements remain unchanged.

Because values are assigned to the specified array elements and not assigned beginning with the first element, the READ statement can assign new values and not alter unspecified elements. For example, the following line assigns values to MYRAY elements 5-7; the unspecified elements remain unchanged:

```
$SAM2 MYRAY(5) = 9, 85, 60 $END
```

Namelist-directed formatting follows the following rules for list-directed input:

- Do not use spaces or tabs in groupname definitions. In value assignments, the entity name cannot contain spaces or tabs except within a subscript or substring specifier, where they are acceptable within the parentheses.
- The groupname and each entity must be contained within a single record.
- When assigning values, you can precede and follow the equal sign with any number of tabs and spaces.

- *Character constants are enclosed in apostrophes. If you want an apostrophe to appear as part of the character string, use two consecutive apostrophes without an intervening blank or end record.*
- *You cannot use Hollerith, octal, or hexadecimal constants.*
- *Character constants can span record boundaries. Normally, the end of a record in namelist input is a space character. If the end of record occurs within a character constant, however, the end of record is ignored; the last character of the previous record is followed by the first character of the next record.*
- *For fixed record length files,* NAMELIST *reads and writes records of a fixed length.*

List-directed output

The format of list-directed output is defined by the data type of the I/O list items except that *r** is not used. Also, neither double nor single quotation marks are output for character constants. Table 46 shows the default output forms that the list-directed WRITE statement generates for each data type.

Table 46
List-directed output formats

Data type	Output format
*LOGICAL*1*	*L2*
*LOGICAL*2*	*L2*
LOGICAL*4	L2
*LOGICAL*8*	*L2*
*INTEGER*1*	*I5*
*INTEGER*2*	*I7*
INTEGER*4	I12
*INTEGER*8*	*I22*
REAL	1PG15.7E2
REAL*8	1PG24.15E3
*REAL*16*	*1PG43.33E4*
COMPLEX	1X,'(',1PG14.7E2,',',1PG14.7E2,')'
*COMPLEX*16*	*1X,'(',1PG23.15E3,',',1PG23.15E3,')'*
CHARACTER	A*n*, where *n* represents the character expression length

Namelist-directed output

The format of namelist-directed output is defined by the data type of the list entities in the corresponding `NAMELIST` *statement. When you use a namelist-directed* `WRITE` *statement, the order of data output is specified by the sequence in which namelist entities are defined in the* `NAMELIST` *statement. For example, assume a program with the following statements:*

```
LOGICAL L4
INTEGER I4
REAL R4
COMPLEX C8
CHARACTER*20 CHAR20

NAMELIST /CONTROL/ L4,I4,R4,C8,CHAR20

READ (5,CONTROL)
WRITE (6,CONTROL)

END
```

with the following input:

```
$CONTROL
     L4 = F, I4 = -123213, C8 = (12,2),
     CHAR20='test case',
     R4=3.14159
$END
```

The `WRITE` *statement outputs the following:*

```
$CONTROL
L4          = F,
I4          = ^^^^-123213,
R4          = ^^3.141590^^^^,
C8          = (^^12.00000^^^^,^^2.000000^^^^),
CHAR20      = 'test^case^^^^^^^^^^^'
$END
```

The output for this program segment consists of the current values of all list entities associated with the namelist specifier. You can output a value for an entity that is defined by the `NAMELIST` *statement but is not assigned an input value. (The entity can also be undefined or defined elsewhere in the program.) For instance, if you had an entity,* `count`*, that was*

defined in the NAMELIST statement but received no input, the current value of count would be written as well as the values shown in the example.

As the example illustrates, each value begins on a new line for namelist-directed output. Character values are enclosed in apostrophes. As mentioned above, the data types used are the same as those defined in the NAMELIST statement. The format output follows the same form as that of list-directed output. Although you can use the $ and & characters interchangeably on input, the $ character is always used for output.

Carriage-control characters

The first character of a formatted record transfers to the printer as a carriage-control character. Table 47 shows the characters that provide vertical format control.

Table 47
Vertical format control

Character	Interpretation
^	Advances one line; begins output at beginning of next line.
0	Advances 2 lines; skips 1 line and begins output.
1	Advances to new page; begins output at the top of a new page.
+	Overwrites; begins output at the beginning of the current line and returns to the left margin.
ASCII NUL	Overwrites with no advance; begins output at beginning of the current line and does not return to left margin.
$	*Prompting; begins output at the beginning of the next line and suppresses carriage return at end of line.*

Specifying FORM='PRINT' in the OPEN statement indicates formatted I/O and implies vertical format control for that unit. You can use the fpr utility to interpret the vertical format controls before printing the file.

Subprograms

9

Subprograms are program units that can be invoked from other program units. Subprograms usually perform frequently used sequences of operations for the invoking program unit. Arguments (dummy and actual) of the subprogram are used to transfer information between the subprogram and another program unit. The dummy argument appears in the argument list of a subprogram and the actual argument appears in the argument list of a subprogram reference.

There are two classes of subprograms: `BLOCK DATA` subprograms (refer to Chapter 10, "Block data subprograms") and procedures. Procedure subprograms include function subprograms and subroutine subprograms. These can include both user defined subprograms and those supplied as part of the CONVEX FORTRAN system.

Dummy and actual arguments

Dummy arguments are classified as variables, arrays, or dummy procedures. They are used in statement functions, function subprograms, and subroutine subprograms to indicate the number and types of actual arguments to be transferred. Dummy arguments indicate whether each actual argument is a single value, array of values, procedure, or statement label. You cannot use a dummy argument name in a `DATA`, `EQUIVALENCE`, `INTRINSIC`, `SAVE`, or `COMMON` statement except as a common block name.

Actual arguments, which can be constants, symbolic names of constants, function references, expressions, arrays and array elements, character substrings, alternate return specifiers, or subprogram names, specify the entities that are to be associated with the dummy arguments. The type of each actual argument must agree with the type of its associated dummy argument,

except when the actual argument is a subroutine name or an alternate return specifier. Actual arguments must also agree in order and number with the dummy arguments.

A function or subroutine reference establishes an association between the corresponding dummy and actual arguments. The dummy argument holds the value of the actual argument during execution. For example, using the following statement:

```
SUBROUTINE SAMPLE (R,L)
```

specifies R and L as dummy arguments. When the subroutine:

```
CALL SAMPLE (B,80)
```

executes, the actual arguments (B,80) replace the dummy arguments (R,L). Thus, B replaces R and 80 replaces L. Any value assigned to R is also assigned to B.

The number of elements of a dummy argument used as an array cannot exceed the number of elements in the actual argument. Also, a type CHARACTER dummy argument length must not be larger than the length of the associated actual argument.

Variables as dummy arguments

To associate a dummy argument variable with an actual argument that is a variable, array element, substring, or expression (including a constant), use the variable, array element, substring, or expression as an actual argument and include a dummy argument of the same data type in the subprogram argument list.

You can define the associated dummy argument within the subprogram if the actual argument is a variable name, array element name, or substring name. If the associated actual argument is a constant or constant name, function reference, or an expression, however, it must not be defined within the subprogram. If you pass a constant to a subroutine as an actual parameter and that subroutine attempts to modify the corresponding dummy argument, either by a READ or ASSIGNMENT statement, a segmentation violation occurs because the constants are stored in read-only storage.

Arrays as dummy arguments

If a dummy argument is declared as an array, it can only be associated with an actual argument that is an array or array element of the same type. To pass an array to a subprogram, use the array name as the actual argument. The subprogram must dimension the array to use it. That is, the dummy array must be specified in an array declarator in the subprogram. The declarator has the same format as that for an actual array but with the following differences:

- You cannot use the declarator in a `COMMON` statement. It is, however, permitted in a `DIMENSION` or type statement.
- Integer constant expressions and expressions containing integer constants and variables can be used as upper and lower bounds of array dimensions. These dimensions are considered adjustable, as one or both of the dimension-bound expressions is a variable. The array is called an adjustable array.
- You can use an asterisk (*)to specify the upper bound of the last dimension. In this case, the array is known as an assumed-size array.

Common block elements must not be passed as actual arguments if the called routine, or any routine that it calls, accesses that element from the common block.

Adjustable arrays

Adjustable arrays are used to process arrays of different sizes in a single subprogram. The adjustable array dimensions are determined in the reference to the subprogram.

Each dummy argument in the array declarator must be associated with an actual argument when the subprogram is entered. Any variable used in an adjustable dimension or each `COMMON` variable appearing in a dimension-bound expression must have a defined value when the subprogram is entered. The expressions specifying the adjustable dimensions are evaluated when the subprogram is entered. Argument association is not retained through different references to the subprogram. The bound values are determined each time a subprogram is entered.

In the statement

```
DIMENSION E(I,I), G(5,2*I)
```

`E` and `G` are adjustable arrays.

The size of the adjustable array must be less than or equal to the size of the array of the corresponding actual argument.

Assumed-size arrays

An asterisk is used to specify the upper bound of the last array dimension in an assumed size array declarator. For example:

```
DIMENSION SAM (*)
```

sets the upper bound to assumed-size for a one-dimensional array. If the array has more than one dimension, only the last dimension can be assumed size. For example

```
DIMENSION SAM (1:N,1:*)
```

sets the upper bound for a two-dimensional array.

The assumed-size dummy array name cannot appear

- In an I/O list of a data transfer statement or an implied DO loop without subscripts.
- As an internal unit identifier in an I/O statement.
- As a runtime format identifier in an I/O statement.

The size of the dummy array is the size of the actual argument array when the actual argument is a noncharacter array name. When the actual argument is a noncharacter array element name, however, the size of the array is the array size plus one minus the subscript value.

The size of the dummy array is `INT(`*n* `+ 1 -` *s*`)/`*l* when the actual argument is one of the following:

- A character array name
- Character array element name
- Character array element substring

and

- *s* is the character storage unit of an array where the actual argument begins.
- *n* is the number or character storage units in this array.
- *l* is the length of an element of the dummy array.

If an assumed size dummy array has *n* dimensions, the product of the sizes of the first *n-1* dimensions must not exceed the size of the array.

Character arguments

You can use character values in one or more of the dummy arguments in a subprogram if the actual argument in the calling program unit is of type CHARACTER. Thus, a dummy argument that is a variable name of type CHARACTER can be associated only with an actual argument that is either a character variable, character array element, character substring, or character expression. A dummy argument that is an array name of type CHARACTER can be associated only with an actual argument that is a character array, character array element, or character array element substring.

If the actual argument is a Hollerith constant (for example, 3HSAM), the dummy argument must be of numeric data type. The corresponding dummy argument can have either a numeric or character data type when the actual argument is a character constant ("SAM").

Character argument lengths

The length of the dummy argument must not exceed the length of the actual argument. The subprogram cannot access more characters than are declared for the argument in the calling unit. That is, when the dummy argument is of type CHARACTER, the associated actual argument must be less than or equal to the length of the actual argument. If the length of the dummy argument of type CHARACTER is less than the length of the associated actual argument, only the leftmost characters of the actual argument are associated with the dummy argument.

If you use an assumed-length character argument, it must be a dummy argument. When control transfers to a subprogram, the assumed-length character dummy argument must be associated with a character actual argument. It assumes the length of the corresponding actual argument. Thus, if you specify the dummy argument length as an assumed-length character argument (for example, `*(*)`), the length of the associated actual argument is used.

If the dummy argument is an array, you can specify a length that differs from that of the calling unit. In this case, however, the subprogram cannot access a character beyond the last character reserved by the calling unit for the array. When a dummy argument of type CHARACTER is an array name, the restriction on length is for the entire array and not for each array element.

You can also use a character array dummy argument with an assumed-length. In this case, the length of each element in the dummy argument equals the length of the elements in the actual argument. The assumed length and the array declarator determine the size of the assumed-length character array.

The following example illustrates length specified for the arguments.

Example:

```
PROGRAM SAM
CHARACTER A1*2, A2*6, A3*8
     .
     .
     .
END

SUBROUTINE MYEX(A)
CHARACTER A*6
     .
     .
     .
END
```

Assume the following CALL statements occur in the main program SAM:

```
CALL MYEX (A1)
CALL MYEX (A2)
CALL MYEX (A3)
```

The first statement is invalid because the length of the dummy argument exceeds that of the associated argument; the remaining two statements are valid with the six leftmost characters of A3 being associated with A. If, however, the subprogram MYEX is defined as

```
SUBPROGRAM MYEXAM (A)
CHARACTER A*(*)
     .
     .
     .
END
```

all three of the CALL statements are valid. The length of the dummy argument A is determined by the length of the corresponding actual argument.

Procedures as dummy arguments

A dummy argument is considered a dummy procedure if the dummy procedure name appears in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement and if:

- It is referenced as a function
- It appears in a type statement and EXTERNAL statement
- It is referenced as a subroutine.

When you use a dummy argument that is a dummy procedure, associate it only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

When you use a dummy argument as an external function, or in a type statement and EXTERNAL statement, the associated actual argument must be an intrinsic function, external function, or dummy procedure.

If you use the dummy argument as a procedure name in a function reference and associate it with an intrinsic function, the arguments must agree in order, number, and type with those specified for the intrinsic function.

When you use the dummy argument as a subroutine, the actual argument must be the name of a subroutine or dummy procedure. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, it is not possible to determine whether the symbolic name becomes associated with a function or subroutine by examination of the subprogram alone.

Alternate return arguments

You can use an asterisk as a dummy argument only in the dummy argument list of a SUBROUTINE statement or an ENTRY statement in a subroutine subprogram. When you use the asterisk as a dummy argument, the corresponding actual argument must be an alternate return specifier in the CALL statement.

Example:

```
SUBROUTINE EXAM(D3,*,E2,*)
```

The alternate return argument allows you to return control to any executable labeled statement in the calling program as long as you have included alternate return arguments in the corresponding positions. These actual arguments have the following form:

**label* or *&label*

Functions

A function can be an intrinsic function, a statement function, or an external function (function subprogram), all of which supply a value to the expression. The function is referenced from within another part of the program. When executed, a function has a value and a type. The general form of a function reference is:

func ([*a* [*a*,]...])

where *func* is the symbolic name of the function or dummy procedure being referenced, and *a* is an optional list of actual arguments separated by commas. If you do not include arguments, you must still include the enclosing parentheses.

Intrinsic functions

CONVEX FORTRAN supplies intrinsic functions as a built-in language feature. You can invoke these pre-existing functions by using the function name in any part of a user program; no definition is required.

There are two classes of intrinsic names: generic names and specific names. If you reference a generic intrinsic name, the compiler decides which special intrinsic to invoke based on the type of the actual arguments. When you reference specific names, the arguments to the intrinsic must be of a specific type. For example, the generic intrinsic function, LOG (natural logarithm), can accept arguments of type REAL, DOUBLE PRECISION, *REAL*16*, and COMPLEX, whereas the specific function, DLOG, can only accept a DOUBLE PRECISION argument.

Using a generic name generally simplifies function referencing because you can use the function name with more than one type of argument. You must use the appropriate specific name whenever the intrinsic function name is used as an actual

argument in a subprogram. For either generic or specific functions that require multiple arguments, all arguments must be of the same data type. The compiler does not convert incorrectly typed arguments.

Use of the `IMPLICIT` statement does not alter the type of intrinsic functions.

An intrinsic reference has the following form:

```
inf (a [,a] [...] )
```

where *inf* is an intrinsic name and *a* is the argument on which the function operates.

Built-in functions

Built-in functions allow a FORTRAN program to pass arguments to a program that is not written in FORTRAN.

`%REF` and `%VAL` functions

Two built-in functions—`%REF` and `%VAL`—can be used in the argument list of a `CALL` statement or function reference to change the form of the argument. Such a change can be necessary if you must call subprograms written in languages other than FORTRAN, because the actual argument may have to be passed in a form different from that used by FORTRAN. These two functions specify how to pass the argument to the subprogram.

Although you can use these functions in the actual argument list of a `CALL` statement or function reference, you cannot use them in any other context. You need not, however, use these built-in functions when invoking a FORTRAN library procedure or a user-supplied subprogram written in FORTRAN.

The two built-in argument list functions are

- `%REF(a)`*—this function passes the argument by reference.*
- `%VAL(a)`*—this function passes the argument as a 32-bit immediate value; an argument shorter than 32 bits is sign-extended to a 32-bit value.*

where a is an actual argument.

Table 48 shows the FORTRAN argument-passing defaults and the allowed uses of `%REF` and `%VAL`.

Table 48
Built-in functions and defaults for argument lists

Data type	*Default*	Functions allowed	
		%REF	%VAL
Expressions			
LOGICAL (*1,2,4)	REF	Yes	Yes†
LOGICAL*8	REF	Yes	No
INTEGER (*1,2,4)	REF	Yes	Yes†
INTEGER*8	REF	Yes	No
REAL*4	REF	Yes	Yes
REAL*8	REF	Yes	No
REAL*16	REF	Yes	No
COMPLEX	REF	Yes	No
CHARACTER	REF	Yes	No
Hollerith	REF	No	No
Array Name			
Numeric	REF	Yes	No
Character	REF	Yes	No
Procedure Name			
Numeric	REF	Yes	No
Character	REF	Yes	No

†If a logical or integer value occupies less than 32 bits of storage, it is converted to a 32-bit value by sign extension.

%LOC function

The %LOC built-in function computes the internal address of a storage element, as in the following example:

```
IADDR = %LOC(v)
```

*where v is a variable name, array element name, array name, character substring name, or external procedure name, and IADDR is an INTEGER*4 variable.*

*The %LOC built-in function produces an INTEGER*4 value that represents the location of its argument. Addresses from user programs are BYTE addresses and are always negative integer values.*

The use of %LOC is limited to arithmetic expressions.

Statement functions

The statement function is a nonexecutable, user-defined, single-statement procedure. You can reference a statement function only from the program unit in which it is defined. The form is similar to an arithmetic, logical, or character assignment statement. *Statement function definitions must precede the use of the statement function.* The statement function returns a single value to the program. The form of a statement function is

```
func ( [d [,d]...] ) = exp
```

where

func
: is the name of the statement function. The type is defined by the implicit naming convention or by a prior type statement. Do not use the name to identify any other entity in the current program unit except a common block.

d
: represents a variable name called a statement function dummy argument. The dummy argument holds the value of the actual argument during execution. The dummy argument list specifies the order, number, and type of actual argument whose values are used in the function reference. The actual arguments must agree in order, number, and type with the corresponding dummy arguments. (The compiler associates the actual arguments with the dummy arguments of the companion statement definition.)

  Each name in the function definition must be unique and must be of the data type of the actual value that replaces it during the function reference. You can use the dummy argument name to identify a variable of the same type, as a dummy argument in a `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement, or as a common block name.

exp
: is an expression. Each primary of *exp* must be one of the following:

  - A constant or symbolic name of a constant
  - A variable reference
  - An array element reference
  - An intrinsic function reference
  - A statement function that has been previously defined in the current program unit

- An external function reference
- A dummy procedure reference

A statement function is referenced using its function reference as a primary in an expression. The following example illustrates a statement function and the statement function reference.

Example:

```
IAVG(IGR1,IGR2,IGR3) = (IGR1 + IGR2 + IGR3)/3
      .
      .
      .
ISC = IAVG(IOR,IWR,IAP)
```

When the statement function reference executes, all actual arguments that are expressions are evaluated and actual arguments are associated with the corresponding dummy arguments. (The compiler substitutes the values in the actual arguments for the dummy arguments.) Then the processor evaluates the expression (right side of the statement function statement). Conversion occurs, if necessary, of the resulting value to the type of the statement function according to the usual arithmetic assignment rules; or a change occurs, if necessary, in the length of a character expression value according to the usual character assignment rules. The resulting value is available to the expression that contains the function reference.

In a statement function reference, any expressions can be used as actual arguments except array names and character expressions involving concatenation of an operand whose length specification is an asterisk (*) in parentheses, unless the operand is the symbolic name of a constant.

Function subprograms

A function subprogram is a separate program unit that includes a `FUNCTION` statement followed by a series of statements that define the computing procedure. The calling program unit references it; the statements execute, and through a `RETURN` or `END` statement, a single value returns to the function reference in the calling unit. This value is assigned to the function name.

The FUNCTION statement specifies the name of the function, the dummy arguments used by the function, and can indicate the type of the function value. In logical and numeric functions, the FUNCTION statement, *including the CONVEX extension *m* and optional parentheses, is represented by:

```
[typ] FUNCTION nam [*m] [ ( [d [,d]...] ) ]
```

where

typ
: is one of the logical or numeric data-type specifiers.

nam
: is the symbolic name of the function subprogram. If you neither specify *typ* nor declare the *nam* in a later type statement, the name implies the data type of the function.

m
: *is an unsigned, nonzero integer constant specifying the length of the data type. It must be one of the valid length specifiers for the data type given by typ.*

d
: is a dummy argument name that can include variable names, array names, or dummy procedure names. All dummy argument names must match the actual arguments in all references in number, order, and type. The dummy name is local to the program unit and must not appear in a DATA, EQUIVALENCE, INTRINSIC, SAVE, or COMMON statement, except as a common block name.

In character functions, the CHARACTER FUNCTION statement, *with the CONVEX extension of *n* and optional parentheses, has the form:

```
CHARACTER[*n] FUNCTION nam [*n] [([ d [,d]...])]
```

where n is either an unsigned, nonzero integer constant, or a parenthetical asterisk (*) indicating an assumed-length function name. If you specify CHARACTER*(*), the function always assumes the length declared for it in the program unit that invokes it. (An assumed-length character function can have different lengths when invoked by different program units.) If *n* is an integer constant, the value of *n* must agree with the length of the function specified in the program unit that invokes the function. If you do not specify *n*, a length of 1 is assumed. If the

length has already been specified after the keyword CHARACTER, you cannot use the optional-length specification following *nam*. Both *nam* and *d* retain the same definition.

You must begin the function subprogram with the FUNCTION statement. You can include any statements except a BLOCK DATA, SUBROUTINE, PROGRAM, or another FUNCTION statement within the function subprogram. End it with an END statement. Between a FUNCTION and END statement, you can use the specified function name as a variable in an executable statement or in a type statement if you omit *typ*. Include ENTRY statements to provide multiple entry points to the subprogram.

A function specified in a subprogram can reference other subprograms but cannot reference itself, directly or indirectly. It must assign a value to its symbolic name at least once.

Subroutine subprograms

A subroutine subprogram (subroutine) is a program unit that performs a specific, user-defined task or subtask for some other program unit of the program. Subroutines are similar to function subprograms; actual and dummy arguments are handled the same for both. The RETURN statement returns control to the calling program. They differ in that the subroutine names have no type, and no value is associated with the subroutine name. Also, you use a CALL statement, not a function reference, to invoke a subroutine. Within the subroutine, you can specify different points of return to the calling subprogram. The CALL statement has the form:

```
CALL sub [ ( [ a [, a] ... ] ) ]
```

where

sub
: is the name of the subroutine.

a

is the actual argument which can be a constant, variable, expression, array, array element, character substring, alternate return specifier, intrinsic function name, external procedure name, or dummy procedure name. You can use an * or & followed by the label of an executable statement to indicate an alternate return. Do not use a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. If the actual argument is a Hollerith constant, the dummy argument must be of numeric data type.

Using the `CALL` statement invokes the subroutine. Control passes to the first executable statement using any *a* arguments for the subroutine dummy arguments. After the subroutine returns, control returns to the statement in the calling unit that follows the `CALL` statement unless you specify an alternate `RETURN` in the subroutine.

You must begin a subroutine with a `SUBROUTINE` statement and end it with an `END` statement. It specifies the name of the subroutine and the arguments used by the subroutine. The `SUBROUTINE` statement has the following form:

```
SUBROUTINE name [ ( [d [, d]...] ) ]
```

where

name

is the symbolic name of the subroutine. Because the subroutine has no data type, you need not apply the naming rules. Because the name is global, do not use it for any other purpose in the program.

d

represents a dummy argument list consisting of a variable name, array name, dummy procedure name, or, if the subroutine uses alternate returns, an asterisk (*). Separate the argument list items with commas. The argument list can be empty. In this case, use of parentheses is optional; for example, either `SUBROUTINE EXAM` or `SUBROUTINE EXAM()` is acceptable.

If you indicate the dummy argument as *, be sure that the corresponding actual argument in the calling unit is also an * or & followed by the label of an executable statement within the

calling unit. You can specify an alternate return in the RETURN statement by giving the position of this asterisk among other asterisks in the dummy argument.

You must specify an actual argument for each dummy argument in the SUBROUTINE statement of the called subroutine. If you use a variable, array element, or array as the actual argument, the data type must match that of the dummy argument. If the argument is the name of a subprogram, you must declare this name in an EXTERNAL statement in this program unit.

The ENTRY statement can be used to specify multiple entry points for subroutines.

ENTRY statement

You can use the nonexecutable ENTRY statement to specify alternative entry points into a function or subroutine subprogram. You can reference an ENTRY from any program unit except the subprogram that contains it. Use a function reference for ENTRY in a function; use CALL for ENTRY in a subroutine. You can place an ENTRY anywhere between the initial FUNCTION or SUBROUTINE statement and the END statement, but you must not place it within a block IF or the range of a DO loop.

The form of the ENTRY statement is

```
ENTRY nam [ ( [d [, d] ... ] ) ]
```

where

nam
: is the symbolic name of the entry point representing either a subroutine name in a subroutine or an external function name in a function subprogram. When entry is in a function, *nam* has a data type that you can imply or specify. If you use a type statement, it can appear before or after the ENTRY statement. For entry in a subroutine, however, there is no data type restriction.

d
: is a variable name, array name, or dummy procedure. You can use an asterisk (*) as an alternate return only if the entry is in a subroutine. You can omit the parentheses for an empty argument list in a subroutine entry, but the parentheses must always be included in a function entry and in the entry reference.

You can use dummy arguments in ENTRY statements that differ in order, number, type, and name from the dummy arguments you use in the FUNCTION, SUBROUTINE, or other ENTRY statements in the same subprogram. Each reference to a function or subroutine, however, must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

If a dummy argument is not currently associated with an actual argument, it is undefined. The association between actual dummy arguments is not retained between references.

An entry name or the name of the function subprogram defines all associated names of the same data type. All entry names within a function subprogram are associated with the name of the function subprogram. You can use function and entry names of different data types.

You must define the variable whose name is used to reference the function before a RETURN or END statement is executed in the subprogram. You don't need to use associated variables of the same type unless the function is type CHARACTER; but an associated variable of different type must not become defined within the subprogram.

RETURN Statement

The RETURN statement is used to return from a subroutine subprogram to one of several alternate points in the calling program unit. You can use none, one, or more than one RETURN statement in a subroutine. Use the alternate return only with subroutine subprograms, not function subprograms. You can, however, use the RETURN statement without an alternate specifier, in either a function or subroutine subprogram.

The statement has the following form:

```
RETURN [e]
```

where *e* is an optional integer expression that specifies an alternate statement in the calling program is to receive control. *The system converts the value type to integer if necessary.* The *e* represents the number, such as RETURN 2, of the corresponding asterisk, among other asterisks, in the dummy argument list of the subroutine. The alternate return specifier has the form of an asterisk or ampersand followed by the label of an executable statement (for example, *30 or &30).

Example:

```
     .
     CALL EXAM(D,*30,E,*40)
     .
     .
     .
30          !RETURN 1 goes here.
     .
     .
     .
40          !RETURN 2 goes here.
     .
     .
     .
     END
SUBROUTINE EXAM(D3,*,E2,*)
     .
     .
     .
RETURN      !Returns after the CALL statement.
     .
     .
     .
RETURN 1    !Returns to 30.
     .
     .
     .
RETURN 2    !Returns to 40.
     .
     .
     .
END
```

In the preceding example, RETURN 1 indicates control transfers to the statement at line 30; RETURN 2 indicates the alternate return transfers control to the statement at line 40. RETURN indicates control transfers to the statement immediately after the CALL statement.

If you do not specify a RETURN, the END statement has the same effect as the RETURN.

Use of the alternate RETURN statement allows you to return control to any labeled statement in the calling program whose label you specify as an alternate return specifier to the

subprogram. If *e* is less than 1 or greater than the total number of asterisks appearing in the dummy argument list, control returns as for a normal RETURN (without specifier).

When a RETURN or END statement executes, the subprogram ends the association between the dummy arguments and the current actual arguments. If the `-re` compiler option is specified, all local data that did not appear in a SAVE statement becomes undefined.

When used within a function, RETURN transfers control to the function reference in the calling unit and returns the function value. In a subroutine, RETURN transfers control to the statement following the CALL statement in the calling program unit.

Block data subprogram 10

A block data subprogram provides initial values for variables and array elements in named common blocks. A block data subprogram must not contain any executable statements.

A block data subprogram begins with a BLOCK DATA statement and ends with an END statement. You can use only one BLOCK DATA statement in a subprogram, but you can use more than one block data subprogram in the program units that constitute the executable program. The statement has the form:

```
BLOCK DATA [name]
```

where *name* is an optional symbolic name for the block data subprogram in which the BLOCK DATA statement appears. Do not assign the BLOCK DATA the same name as that of an external procedure, main program, common block, or other block data subprogram, or any local name in the block data subprogram.

You can use only these specification statements between the BLOCK DATA and END statements: COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and any of the type-declaration statements. Your block data subprogram must contain at least one COMMON statement and one DATA statement.

You must specify all entities having storage units in the common block, but you are not required to initialize all of the values. Be sure to provide specifications to establish the entire block.

Example:

```
BLOCK DATA MYBLOK
COMMON/EX/A,B,C
COMMON/CAT/LIST(100)
DATA A/3.5/,LIST/(100*5)
END
```

Intrinsics and commonly used library routines

A

This appendix lists CONVEX FORTRAN's generic and specific intrinsics, as well as its commonly used library routines.

Generic and specific intrinsics

Table 49 lists generic and specific intrinsics. Numbers in the first and second column of Table 49 refer to notes following the table.

Table 49
Generic and specific intrinsics

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
SQRT	SQRT	Square root $a^{1/2}$	1	REAL*4	REAL*4
1	DSQRT			REAL*8	REAL*8
	QSQRT			*REAL*16*	*REAL*16*
	CSQRT			COMPLEX*8	COMPLEX*8
	CDSQRT			*COMPLEX*16*	*COMPLEX*16*
LOG	ALOG	Natural log $\log_e a$	1	REAL*4	REAL*4
2	DLOG			REAL*8	REAL*8
	QLOG			*REAL*16*	*REAL*16*
	CLOG			COMPLEX*8	COMPLEX*8
	CDLOG			*COMPLEX*16*	*COMPLEX*16*
LOG10	ALOG10	Common log $\log_{10} a$	1	REAL*4	REAL*4
2	DLOG10			REAL*8	REAL*8
	QLOG10			*REAL*16*	*REAL*16*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
EXP	EXP DEXP *QEXP* CEXP *CDEXP*	Exponential e^a	1	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*
SIN 3	SIN DSIN *QSIN* CSIN *CDSIN*	Sine sin a	1	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*
SIND 3	*SIND* *DSIND* *QSIND*	*Sine (degree)* *sin* a°	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
COS 3	COS DCOS *QCOS* CCOS *CDCOS*	Cosine cos a	1	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8 *REAL*16* COMPLEX*8 *COMPLEX*16*
COSD 3	*COSD* *DCOSD* *QCOSD*	*Cosine (degree)* *cos* a°	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
TAN 3	TAN DTAN *QTAN*	Tangent tan a	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
TAND 3	*TAND* *DTAND* *QTAND*	*Tangent (degree)* *tan* a°	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ASIN 4,5	ASIN DASIN *QASIN*	Arcsine arcsin a	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ASIND 2,4,5	*ASIND* *DASIND* *QASIND*	*Arcsine (degree)* *arcsin* a	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
ACOS 4,5	ACOS DACOS *QACOS*	Arccosine arccos *a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ACOSD 2,4,5	*ACOSD* *DACOSD* *QACOSD*	*Arccosine (degree)* *arccos a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ATAN 5	ATAN DATAN *QATAN*	Arctangent arctan *a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ATAND 2,5	*ATAND* *DATAND* *QATAND*	*Arctangent (degree)* *arctan a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ATAN2 5,6	ATAN2 DATAN2 *QATAN2*	Arctangent (two arguments) arctan a_1/a_2	2	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
ATAN2D 2,5,7	*ATAN2D* *DATAN2D* *QATAN2D*	*Arctangent (two degree arguments)* *Arctan* a_1/a_2	2	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
SINH	SINH DSINH *QSINH*	Hyperbolic Sine sinh *a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
COSH	COSH DCOSH *QCOSH*	Hyperbolic Cosine cosh *a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*
TANH	TANH DTANH *QTANH*	Hyperbolic Tangent tanh *a*	1	REAL*4 REAL*8 *REAL*16*	REAL*4 REAL*8 *REAL*16*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
ABS	IIABS	Absolute value	1	INTEGER*2	INTEGER*2
8	JIABS	|a|		INTEGER*4	INTEGER*4
	KIABS			INTEGER*8	INTEGER*8
	ABS			REAL*4	REAL*4
	DABS			REAL*8	REAL*8
	QABS			REAL*16	REAL*16
	CABS			COMPLEX*8	REAL*4
	CDABS			COMPLEX*16	REAL*8
IABS	IIABS	Absolute value	1	INTEGER*2	INTEGER*2
8	JIABS	|a|		INTEGER*4	INTEGER*4
	KIABS			INTEGER*8	INTEGER*8
INT	IINT	Truncation	1	REAL*4	INTEGER*2
9,14	JINT	[a]		REAL*4	INTEGER*4
15	KINT			REAL*4	INTEGER*8
	IIDINT			REAL*8	INTEGER*2
	JIDINT			REAL*8	INTEGER*4
	KIDINT			REAL*8	INTEGER*8
	IIQINT			REAL*16	INTEGER*2
	JIQINT			REAL*16	INTEGER*4
	KIQINT			REAL*16	INTEGER*8
INT				COMPLEX*8	INTEGER*2
19				COMPLEX*8	INTEGER*4
				COMPLEX*8	INTEGER*8
				COMPLEX*16	INTEGER*2
				COMPLEX*16	INTEGER*4
				COMPLEX*16	INTEGER*8
10,14	INT1	Conversion to INTEGER	1	Any numeric	INTEGER*1
	INT2			Any numeric	INTEGER*2
	INT4			Any numeric	INTEGER*4
	INT8			Any numeric	INTEGER*8
IDINT	IIDINT	Truncation	1	REAL*8	INTEGER*2
9,14	JIDINT	[a]		REAL*8	INTEGER*4
15	KIDINT			REAL*8	INTEGER*8

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
IQINT	*IIQINT*	*Truncation*	1	*REAL*16*	*INTEGER*2*
9,14	*JIQINT*	*[a]*		*REAL*16*	INTEGER*4
15	*KIQINT*			*REAL*16*	*INTEGER*8*
AINT	AINT	*Truncation*	1	REAL*4	REAL*4
	DINT	*[a]*		REAL*8	REAL*8
	QINT			*REAL*16*	*REAL*16*
NINT	*ININT*	*Nearest integer*	1	REAL*4	*INTEGER*2*
9,14	*JNINT*	*[a+.5*sign(a)]*		REAL*4	INTEGER*4
15	*KNINT*			REAL*4	*INTEGER*8*
	IIDNNT			REAL*8	*INTEGER*2*
	JIDNNT			REAL*8	INTEGER*4
	KIDNNT			REAL*8	*INTEGER*8*
	IIQNNT			*REAL*16*	*INTEGER*2*
	JIQNNT			*REAL*16*	INTEGER*4
	KIQNNT			*REAL*16*	*INTEGER*8*
IDNINT	*IIDNNT*	*Nearest integer*	1	REAL*8	*INTEGER*2*
9,14	*JIDNNT*	*[a+.5*sign(a)]*		REAL*8	INTEGER*4
15	*KIDNNT*			REAL*8	*INTEGER*8*
IQNINT	*IIQNNT*	*Nearest integer*	1	*REAL*16*	*INTEGER*2*
9,14	*JIQNNT*	*[a+.5*sign(a)]*		*REAL*16*	INTEGER*4
15	*KIQNNT*			*REAL*16*	*INTEGER*8*
ANINT	ANINT	Nearest integer	1	REAL*4	REAL*4
9,15	DNINT	[*a*+.5*sign(*a*)]		REAL*8	REAL*8
	QNINT			*REAL*16*	*REAL*16*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
ZEXT	*IZEXT*	*Zero-extend a*	1	*LOGICAL*1*	*INTEGER*2*
14,15	19			*LOGICAL*2*	*INTEGER*2*
	19			*INTEGER*2*	*INTEGER*2*
	JZEXT			*LOGICAL*1*	INTEGER*4
	19			*LOGICAL*2*	INTEGER*4
	19			LOGICAL*4	INTEGER*4
	19			INTEGER*2	INTEGER*4
	19			INTEGER*4	INTEGER*4
	KZEXT			*LOGICAL*1*	*INTEGER*8*
	19			*LOGICAL*2*	*INTEGER*8*
	19			LOGICAL*4	*INTEGER*8*
	19			LOGICAL*8	*INTEGER*8*
	19			*INTEGER*2*	*INTEGER*8*
	19			INTEGER*4	*INTEGER*8*
	19			*INTEGER*8*	*INTEGER*8*
REAL	*FLOATI*	Convert *a* to *REAL*4*	1	*INTEGER*2*	REAL*4
10	*FLOATJ*			INTEGER*4	REAL*4
	FLOATK			*INTEGER*8*	REAL*4
	19			REAL*4	REAL*4
	SNGL			REAL*8	REAL*4
	19			COMPLEX*8	REAL*4
	19			*COMPLEX*16*	REAL*4
	SNGLQ			*REAL*16*	REAL*4
DBLE	19	Convert *a* to REAL*8	1	*INTEGER*2*	REAL*8
10	19			INTEGER*4	REAL*8
	19			*INTEGER*8*	REAL*8
	19			REAL*4	REAL*8
	19			REAL*8	REAL*8
	19			COMPLEX*8	REAL*8
	19			*COMPLEX*16*	REAL*8
	DBLEQ			*REAL*16*	REAL*8

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
QEXT		*Convert a to REAL*16*	1	*INTEGER*2*	*REAL*16*
19				INTEGER*4	*REAL*16*
				*INTEGER*8*	*REAL*16*
				REAL*4	*REAL*16*
	QEXTD			REAL*8	*REAL*16*
	19			*REAL*16*	*REAL*16*
	19			COMPLEX*8	*REAL*16*
	19			*COMPLEX*16*	*REAL*16*
IFIX	*IIFIX*	*Fix a (REAL*4-to-INTEGER conversion)*	1	REAL*4	*INTEGER*2*
10,14	*JIFIX*			REAL*4	INTEGER*4
15	*KIFIX*			REAL*4	*INTEGER*8*
FLOAT	*FLOATI*	*Float a (INTEGER-to-REAL*4 conversion)*	1	*INTEGER*2*	REAL*4
10	*FLOATJ*			INTEGER*4	REAL*4
	FLOATK			*INTEGER*8*	REAL*4
DFLOAT	*DFLOTI*	*Float a (INTEGER-to-REAL*8 conversion)*	1	*INTEGER*2*	REAL*8
10	*DFLOTJ*			INTEGER*4	REAL*8
	DFLOTK			*INTEGER*8*	REAL*8
QFLOAT	*QFLOTI*	*Float a (INTEGER-to-REAL*16 conversion)*	1	*INTEGER*2*	*REAL*16*
10	*QFLOTJ*			INTEGER*4	*REAL*16*
	QFLOTK			*INTEGER*8*	*REAL*16*
CMPLX		Convert a_1 to COMPLEX*8 or convert a_1 and a_2 to COMPLEX*8	1,2	*INTEGER*2*	COMPLEX*8
19			1,2	*INTEGER*2*	COMPLEX*8
			1,2	INTEGER*4	COMPLEX*8
			1,2	REAL*4	COMPLEX*8
			1,2	REAL*8	COMPLEX*8
			1	COMPLEX*8	COMPLEX*8
			1	*COMPLEX*16*	COMPLEX*8
DCMPLX		*Convert a_1 to COMPLEX*8 or convert a_1 and a_2 to COMPLEX*8*	1,2	*INTEGER*2*	*COMPLEX*16*
11,15			1,2	INTEGER*4	*COMPLEX*16*
19			1,2	*INTEGER*8*	*COMPLEX*16*
			1,2	REAL*4	*COMPLEX*16*
			1,2	REAL*8	*COMPLEX*16*
			1	COMPLEX*8	*COMPLEX*16*
			1	*COMPLEX*16*	*COMPLEX*16*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
	REAL *DREAL*	Real part of complex	1	COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8
	AIMAG *DIMAG*	Imaginary part of complex	1	COMPLEX*8 *COMPLEX*16*	REAL*4 REAL*8
CONJG	CONJG *DCONJG*	Complex conjugate (if $a=(x,y)$ then CONJG$(a)=(x,-y)$)	1	COMPLEX*8 *COMPLEX*16*	COMPLEX*8 *COMPLEX*16*
	DPROD	REAL*8 product of REAL*4 a_1*a_2	2	REAL*4	REAL*8
MAX 12,15	*IMAX0* *JMAX0* *KMAX0* *AMAX1* *DMAX1* *IIDMAX1* *JIDMAX1* *KIDMAX1* *QMAX1*	Maximum max(a_1, a_2,...a_n)	n	*INTEGER*2* INTEGER*4 INTEGER*8 REAL*4 REAL*8 REAL*8 REAL*8 REAL*8 *REAL*16*	*INTEGER*2* INTEGER*4 INTEGER*8 REAL*4 REAL*8 *INTEGER*2* INTEGER*4 INTEGER*8 *REAL*16*
MAX0 12,15	*IMAX0* *JMAX0* *KMAX0*	*Maximum max(a_1,a_2,...a_n)*	n	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
MAX1 12,14 15	*IMAX1* *JMAX1* *KMAX1*	*Maximum max(a_1,a_2,...a_n)*	n	REAL*4 REAL*4 REAL*4	*INTEGER*2* INTEGER*4 *INTEGER*8*
AMAX0 12,15	*AIMAX0* *AJMAX0* *AKMAX0*	*Maximum max(a_1,a_2,...a_n)*	n	*INTEGER*2* INTEGER*4 *INTEGER*8*	REAL*4 REAL*4 REAL*4

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
MIN	IMIN0	Minimum min(a_1,a_2,...a_n)	n	INTEGER*2	INTEGER*2
13, 15	JMIN0			INTEGER*4	INTEGER*4
	KMIN0			INTEGER*8	INTEGER*8
	AMIN1			REAL*4	REAL*4
	DMIN1			REAL*8	REAL*8
	IIDMIN1			REAL*8	INTEGER*2
	JIDMIN1			REAL*8	INTEGER*4
	KIDMIN1			REAL*8	INTEGER*8
	QMIN1			REAL*16	REAL*16
MIN0	IMIN0	Minimum min(a_1,a_2,...a_n)	n	INTEGER*2	INTEGER*2
13,15	JMIN0			INTEGER*4	INTEGER*4
	KMIN0			INTEGER*8	INTEGER*8
MIN1	IMIN1	Minimum min(a_1,a_2,...a_n)	n	REAL*4	INTEGER*2
13,14	JMIN1			REAL*4	INTEGER*4
15	KMINI			REAL*4	INTEGER*8
AMIN0	AIMIN0	Minimum min(a_1,a_2,...a_n)	n	INTEGER*2	REAL*4
	AJMIN0			INTEGER*4 5	REAL*4
DIM	IIDIM	Positive difference a_1- (min(a_1, a_2))	2	INTEGER*2	INTEGER*2
	JIDIM			INTEGER*4	INTEGER*4
	KIDIM			INTEGER*8	INTEGER*8
	DIM			REAL*4	REAL*4
	DDIM			REAL*8	REAL*8
	QDIM			REAL*16	REAL*16
IDIM	IIDIM	Positive difference a_1- (min(a_1, a_2))	2	INTEGER*2	INTEGER*2
	JIDIM			INTEGER*4	INTEGER*4
	KIDIM			INTEGER*8	INTEGER*8
MOD	IMOD	Remainder $a_1-a_2*[a_1/a_2]$ (Returns the remainder when the first argument is divided by the second)	2	INTEGER*2	INTEGER*2
	JMOD			INTEGER*4	INTEGER*4
	KMOD			INTEGER*8	INTEGER*8
	AMOD			REAL*4	REAL*4
	DMOD			REAL*8	REAL*8
	QMOD			REAL*16	REAL*16

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
SIGN	*IISIGN*	Transfer of sign	2	*INTEGER*2*	*INTEGER*2*
	JISIGN	$\lvert a_1 \rvert$ if $a_2 \geq 0$		INTEGER*4	INTEGER*4
	KISIGN	$-\lvert a_1 \rvert$ if $a_2 < 0$		*INTEGER*8*	*INTEGER*8*
	SIGN			REAL*4	REAL*4
	DSIGN			REAL*8	REAL*8
	QSIGN			*REAL*16*	*REAL*16*
ISIGN	*IISIGN*	Transfer of sign	2	*INTEGER*2*	*INTEGER*2*
	JISIGN	$\lvert a_1 \rvert$ sign a_2		INTEGER*4	INTEGER*4
	KISIGN			*INTEGER*8*	*INTEGER*8*
IAND	*IIAND*	*Bitwise AND (performs a logical AND on corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIAND			INTEGER*4	INTEGER*4
	KIAND			*INTEGER*8*	*INTEGER*8*
IOR	*IIOR*	*Bitwise OR (performs an inclusive OR on corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIOR			INTEGER*4	INTEGER*4
	KIOR			*INTEGER*8*	*INTEGER*8*
IEOR	*IIEOR*	*Bitwise XOR (exclusively ORs corresponding bits)*	2	*INTEGER*2*	*INTEGER*2*
	JIEOR			INTEGER*4	INTEGER*4
	KIEOR			*INTEGER*8*	*INTEGER*8*
NOT	*INOT*	*Bitwise Complement (complements each bit)*	1	*INTEGER*2*	*INTEGER*2*
	JNOT			INTEGER*4	INTEGER*4
	KNOT			*INTEGER*8*	*INTEGER*8*
ISHFT† 16	IISHFT	Bitwise shift (a_1 logically shifted a_2 bits—positive a_2 argument shifts left; negative, right)	2	*INTEGER*2*	*INTEGER*2*
	JISHFT			INTEGER*4	INTEGER*4
	KISHFT			*INTEGER*8*	*INTEGER*8*
IBITS† 17	*IIBITS*	Bit extraction (extracts bits a_2 through a_2+a_3-1 from a_1)	3	*INTEGER*2*	*INTEGER*2*
	JIBITS			INTEGER*4	INTEGER*4
	KIBITS			*INTEGER*8*	*INTEGER*8*
IBSET†	*IIBSET*	Bit set (returns the value of a_1 with bit a_2 of a_1 set to 1)	2	*INTEGER*2*	*INTEGER*2*
	JIBSET			INTEGER*4	INTEGER*4
	KIBSET			*INTEGER*8*	*INTEGER*8*
BTEST†	*BITEST*	Bit test (returns TRUE if bit a_2 of argument a_1 equals 1)	2	*INTEGER*2*	*LOGICAL*2*
	BJTEST			INTEGER*4	LOGICAL*4
	BKTEST			*INTEGER*8*	*LOGICAL*8*

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
IBCLR†	*IIBCLR* JIBCLR *KIBCLR*	Bit Clear (returns the value of a_1 with bits a_2 of a_1 set to 0)	2	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
ISHFTC†	*IISHFTC* JISHFTC *KISHFTC*	Bitwise circular shift (circularly shifts rightmost a_3 bits of argument a_1 by a_2)	3	*INTEGER*2* INTEGER*4 *INTEGER*8*	*INTEGER*2* INTEGER*4 *INTEGER*8*
15	LEN	Length (returns length of the character expression)	1	CHARACTER	INTEGER*4
15	INDEX	Index(c_1,c_2) (returns the position of the substring c_2 in the character expression c_1)	2	CHARACTER	INTEGER*4
15 19	CHAR	Character (returns a character that has the ASCII value specified by the argument)	1	*LOGICAL*1* *INTEGER*2* INTEGER*4 *INTEGER*8*	CHARACTER CHARACTER CHARACTER CHARACTER
15	ICHAR	ASCII value (returns the ASCII value of the argument, which must be a character expression of length 1)	1	CHARACTER	INTEGER*4
	LLT LLE LGT LGE	Character relationals (ASCII collating sequence)	2	CHARACTER CHARACTER CHARACTER CHARACTER	LOGICAL*4 LOGICAL*4 LOGICAL*4 LOGICAL*4
DOT_PRODUCT *18*		*Dot product of two rank-one arrays*	2	*Varies*††	*Varies*††
MATMUL *18*		*Matrix multiply*	2	*Varies*††	*Varies*††
ALL *18*		*True if all array elements along optional dimension are true*	1,2	*Varies*††	*Varies*††
ANY *18*		*True if any array element along optional dimension is true*	1,2	*Varies*††	*Varies*††
COUNT *18*		*Counts number of true elements along optional dimension in array*	1,2	*Varies*††	*Varies*††

Table 49
(continued)

Intrinsics		Function	No. args	Argument type	Result type
Generic	Specific				
MAXVAL *18*		*Maximum value along optional dimension using optional mask*	1-3	*Varies*††	*Varies*††
MINVAL *18*		*Minimum value along optional dimension using optional mask*	1-3	*Varies*††	*Varies*††
PRODUCT *18*		*Product of array elements along optional dimension using optional mask*	1-3	*Varies*††	*Varies*††
SUM *18*		*Sum all the elements of an an array along optional dimension, with an optional mask*	1-3	*Varies*††	*Varies*††
MERGE *18*		*Merge under mask*	3	*Varies*††	*Varies*††
PACK *18*		*Pack an array into an array of rank one under mask, optionally sizing the resulting array*	2,3	*Varies*††	*Varies*††
SPREAD *18*		*Replicates array by adding a dimension*	3	*Varies*††	*Varies*††
UNPACK *18*		*Unpack an array of rank one into an array under a mask*	3	*Varies*††	*Varies*††
TRANSPOSE *18*		*Transpose of an array of rank two*	1	*Varies*††	*Varies*††
MAXLOC *18*		*Location of a maximum value in an array under an optional mask*	2,3	*Varies*††	*Varies*††
MINLOC *18*		*Location of a minimum value in an array under an optional mask*	2,3	*Varies*††	*Varies*††

†Arguments after the first are converted to the type of the first argument.
†† The argument list to the corresponding library routines is variable for these Fortran 90 intrinsics. The types of the arguments to these functions are not predetermined.

Notes:

1. The SQRT, DSQRT, or QSQRT argument must be greater than or equal to zero. The result is the principal value where the real part is greater than or equal to zero. If the real part is zero, the result is the principal value and the imaginary part greater than or equal to zero.

2. The ALOG, DLOG, QLOG, ALOG10, DLOG10, QLOG10, ATAND, ATAN2D, ASIND, DASIND, QASIND, ACOSD, DACOSD, or QACOSD argument, must be greater than zero. The CLOG or CDLOG argument cannot be (0.,0.).

3. The SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, or QTAN argument must be in radians, and is treated as modulo 2π. The SIND, COSD, or TAND argument must be in degrees; the argument is treated as modulo 360. The value of the sine and cosine functions for very large arguments is not meaningful because of the accuracy of the argument reduction. For single precision, the maximum value is $\pi * 2^{23}$; for double precision, the maximum value is $\pi * 2^{52}$; for quadruple precision, the maximum value is $\pi * 2^{112}$. If the argument exceeds the maximum, it is replaced with 0 and evaluation continues.

4. The absolute value of the ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ASIND, DASIND, QASIND, ACOSD, DACOSD, or QACOSD argument must be less than or equal to 1.

5. The result of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ATAN, DATAN, QATAN, ATAN2, DATAN2, or QATAN2 is in radians, and that of ASIND, DASIND, QASIND, ACOSD, DACOSD, QACOSD, ATAND, DATAND, QATAND, ATAN2D, DATAN2D, or QATAN2D is in degrees.

6. If the value of the first ATAN2, DATAN2, or QATAN2 argument is positive, the result is positive; if it is zero, the result is zero if the second argument is positive, and π if the second argument is negative. A negative value for the first argument determines a negative result. A zero value for the second argument results in the absolute value of $\pi/2$. Both arguments cannot have the value zero. The range of the result for ATAN2, DATAN2, or QATAN2 is $-\pi < \text{result} \leq \pi$.

7. If the value of the first ATAN2D, DATAN2D, or QATAN2D argument is positive, the result is positive. If it is zero, the result is zero if the second argument is positive, and

180 degrees if the second argument is negative. A negative value for the first argument means the result is negative. A zero value for the second argument results in the absolute value of 90 degrees. Both arguments cannot be zero. The range of the result for `ATAN2`, `DTAN2D`, or `QATAN2D` is: -180 degrees < result ≤ 180 degrees.

8. The absolute value of a complex number, `(X,Y)`, is the real value: `(X**2 + Y**2)**1/2`.

9. Define [x] as the largest integer whose magnitude is not greater than the magnitude of x, and whose sign matches that of x. For example [5.7] equals 5 and [-5.7] equals -5.

10. Functions used to convert one data type to another have the same effect as the implied conversion in assignment statements. The functions `REAL` with a real argument, `DBLE` with a double-precision argument, and `INT` with an integer argument return the value of the argument without conversion.

11. If `CMPLX` or `DCMPLX` has only one argument, the argument converts into the real part of a complex value, and zero is assigned to the imaginary part. If there are two arguments (not complex), conversion of the first argument into the real part of the value, and the second argument into the imaginary part, produces a complex value.

12. This function causes the return of the maximum value from the argument list; there must be at least two arguments.

13. This function causes the return of the minimum value from the argument list; there must be at least two arguments.

14. This function converts to the default `INTEGER` type.

15. The `INT`, `IDINT`, `IQINT`, `NINT`, `IDNINT`, `IQNINT`, `IFIX`, `MAX1`, `MINI`, and `ZEXT` functions return INTEGER*4 values if the `/I4` (COVUE only) or `-i4` flag is in effect, *INTEGER*2* values if the */NOI4* (COVUE only) or *-i2* flag is in effect, or *INTEGER*8* values if the *- i8*, *-p8*, or *-pd8* flag is in effect.

16. These functions shift binary patterns—positive, left (a_1 is >0) and negative, right ($a_2 < 0$). Since `ISHFT` indicates a logical shift, the bits shifted out of one end are lost and zeros are shifted in at the other end. As `ISHFTC` specifies a circular shift, the bits shifted out at one end are shifted back in at the other end.

17. In CONVEX FORTRAN, bits within a word are numbered from right to left. For example, in `REAL*4` data, the rightmost (least significant) bit is 0; the leftmost (most significant) bit is 31.

    Bit extraction proceeds from right to left; thus, a_2 is the rightmost bit extracted and a_2+a_3-1 is the leftmost bit. The extracted field is shifted right before being placed into the receiving field. If the extracted field is shorter than the receiving field, zeros are filled in from the left.

    The following example illustrates bit extraction.

```
      PROGRAM TEST
      INTEGER*4 I1, J1, K1, L1
      I1 = 'FFFFFFFF'X

      J1 = IBITS (I1,1,4)
      K1 = IBITS (I1,29,4)
      L1 = IBITS (I1,16,6)

      WRITE (6,100) 'I1 = ',I1
      WRITE (6,100) 'J1 = ',J1
      WRITE (6,100) 'K1 = ',K1
      WRITE (6,100) 'L1 = ',L1

  100 FORMAT (A4,Z8)
      END
```

    When the program is executed, the output displays the original bit pattern and the extracted bit fields.

```
I1 =FFFFFFFF
J1 =       F
K1 =       7
L1 =      3F
```

18. These are Fortran 90 intrinsics. They have no user-callable specific functions. For a more detailed explanation of their use, refer to Chapter 2, "FORTRAN statement components."

19. No user-callable specific functions are available for these argument/result pairs.

Commonly used library routines

Commonly used library subroutines and functions are listed below. Functions are denoted as such; all other listed utilities are subroutines. For more information refer to the *CONVEX FORTRAN User's Guide*, Chapter 6, "System Utilities."

`date` (*buf*)
: Returns current date as mm-dd-yy. *buf* is of type `CHAR*9`. This subroutine behaves differently in `-cfc` mode; refer to Appendix G, "Cray FORTRAN compatibility."

`idate` (*i*, *j*, *k*)
: Returns current month (*i*), day (*j*), and year (*k*). *i*, *j*, and *k* are of type `INTEGER*4`.

`errsns` (*fnum*, *rmssts*, *rmsstv*, *iunit*, *condval*)
: Returns information about last runtime error in *fnum*; remaining arguments are not used. All arguments are of type `INTEGER*4`.

`exit` (*status*)
: Ends a process and makes the argument *status* available to the parent process. *status* is of type `INTEGER*4`.

`secnds` (*x*)
: (function) Returns system time minus the value of its argument in seconds. Both *x* and `secnds` (*x*) are of type `REAL*4`.

`time` (*buf*)
: Returns current system time in an ASCII string as hh:mm:ss. *buf* is of type `CHAR*8`.

`ran` (*i*)
: (function) Returns random integer; similar to ConvexOS utility `rand`. Both *i* and `ran` (*i*) are of type `INTEGER*4`.

`mvbits(m, i, len, n, j)`

Transfers *len* bits from position *i* through *i+len-1* of source location *m* to position *j* through *j+len-1* of destination location *n*;
i+len, *j+len* < 32. All arguments are of type `INTEGER*4`.

Preprocessor B

Previous versions of the FORTRAN compiler had a preprocessor. This preprocessor is now 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*

`#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 in `-I` options on the command line, and, finally, in the standard search path. `#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

`#ifndef` *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` statment 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

Certain compiler command line options are associated exclusively with the preprocessor. These options, with the exception of `-fpp`, are being phased out. The options are as follows:

`-D`*name*`[=`*def*`]`
: Defines *name* to the preprocessor as if it had been specified in a `#define` statement. If no definition (*def*) is given, the name is defined as 1.

`-E`
: Runs only the FORTRAN preprocessor on the named FORTRAN programs and sends the result to the standard output file.

`-fpp`
: Runs the FORTRAN preprocessor as the first step of compilation. If this option is not specified, the preprocessor is not used.

`-I`*dir*
: For `#include` files whose names do not begin with "/", the preprocessor searches first in the directory of the file argument, then in directories named in `-I`, then in directories on a standard list. Directories given in `-I` options are searched in the order in which they appear on the command line. No more than eight directories can be specified.

-U*name*

Removes any initial definition of *name*. The only built-in names defined are __LINE__, and __FILE__.

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 non-zero status prevents the driver from compiling the output file. The preprocessor should generate its own error message (on `stderr`) if a non-zero 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.

Example:

```
#!/bin/sh
# mypp -- user defined fortran preprocessor
#
# emulates -dc flag
#
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
```

Compiler directives

C

This appendix briefly describes the directives that are available in CONVEX FORTRAN.

Some 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, vectorization, or parallelization. A directive line has the following format:

C$DIR *directive* [, *directive*]

A directive line begins in column one with the characters C$DIR followed by one or more of the directives described in this appendix. If two or more directives are specified, they are separated by commas. A directive must fit on one line; it cannot be continued. A *directive* can be surrounded by any number of comment lines.

The following directives are supported:

- ASSIGN_LOCK, FREE_LOCK
- BEGIN_ORDER, END_ORDER
- BEGIN_SECTION, END_SECTION
- BEGIN_TASKS, NEXT_TASK, END_TASKS
- FORCE_PARALLEL_EXT
- FORCE_PARALLEL
- FORCE_VECTOR
- MAX_TRIPS
- NO_PARALLEL
- NO_PEEL

- NO_PROMOTE_TEST
- NO_RECURRENCE
- NO_SIDE_EFFECTS
- NO_VECTOR
- PEEL
- PEEL_ALL
- PREFER_PARALLEL_EXT
- PREFER_PARALLEL
- PREFER_VECTOR
- PSTRIP
- PROMOTE_TEST
- PROMOTE_TEST_ALL
- ROW_WISE
- SCALAR
- SELECT
- SYNCH_PARALLEL
- UNROLL
- VSTRIP

Certain combinations of directives are invalid when used within the same program unit or loop and cause the program unit or loop to be rejected by the compiler. Table 50 lists invalid combinations.

Table 50
Restrictions on directive use

The directive...	Cannot be used with...
FORCE_PARALLEL	FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_PARALLEL, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL, VSTRIP
FORCE_PARALLEL_EXT	FORCE_PARALLEL, NO_PARALLEL, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, PSTRIP, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL
FORCE_VECTOR	FORCE_PARALLEL, NO_VECTOR, PREFER_VECTOR, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PSTRIP, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL
NO_PARALLEL	FORCE_PARALLEL, FORCE_PARALLEL_EXT, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PSTRIP, SCALAR, SELECT, SYNCH_PARALLEL, VSTRIP
NO_VECTOR	FORCE_VECTOR, PREFER_VECTOR, SCALAR, SELECT, VSTRIP
PREFER_PARALLEL	FORCE_PARALLEL, FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL, VSTRIP
PREFER_PARALLEL_EXT	FORCE_PARALLEL, FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_PARALLEL, PREFER_PARALLEL, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL
PREFER_VECTOR	FORCE_PARALLEL, FORCE_PARALLEL_EXT, NO_VECTOR, FORCE_VECTOR, PREFER_PARALLEL, PSTRIP, SCALAR, SELECT, SYNCH_PARALLEL, UNROLL
PSTRIP	FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_PARALLEL, PREFER_VECTOR, PREFER_PARALLEL_EXT, SCALAR, SYNCH_PARALLEL, UNROLL, VSTRIP
SCALAR	FORCE_PARALLEL, FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_PARALLEL, NO_VECTOR, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, PSTRIP, SELECT, SYNCH_PARALLEL, VSTRIP
SELECT	FORCE_PARALLEL, FORCE_PARALLEL_EXT, FORCE_VECTOR, NO_VECTOR, NO_PARALLEL, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, SCALAR, UNROLL SYNCH_PARALLEL
UNROLL	FORCE_PARALLEL, FORCE_PARALLEL_EXT, FORCE_VECTOR, PREFER_PARALLEL, PREFER_PARALLEL_EXT, PREFER_VECTOR, PSTRIP, SELECT, SYNCH_PARALLEL, VSTRIP
VSTRIP	FORCE_PARALLEL, NO_PARALLEL, NO_VECTOR, PREFER_PARALLEL, PSTRIP, SCALAR, SYNCH_PARALLEL, UNROLL

The scope of a directive associated with a loop is the loop immediately following the directive and does not include any nested loops.

When using directives on loops, remember that loops can be executed in the following ways:

- Serial
- Vector but not parallel
- Parallel but not vector
- Parallel outer strip and vector inner strip

The following pages describe the individual directives. Where a directive has arguments associated with it, the format of the directive is shown.

ASSIGN_LOCK, FREE_LOCK

The ASSIGN_LOCK directive designates an integer scalar variable or array element to be used as a lock to control ordered and unordered critical regions during parallelization. The FREE_LOCK directive releases a lock previously assigned by an ASSIGN_LOCK directive. The ASSIGN_LOCK and FREE_LOCK directives must be in the same program unit as the loop that is to be parallelized.

The formats of these directives are as follows:

```
ASSIGN_LOCK (lockname [,lockname]...)
FREE_LOCK   (lockname [,lockname]...)
```

where *lockname* is the name of the locking variable being assigned or freed. The locking variable must be declared as either INTEGER*4 or INTEGER*8.

The ASSIGN_LOCK directive initially places the lock in the unlocked state. The lock is set or reset as it is used during execution of critical regions.

Be careful to avoid errors such as freeing a lock before assigning it, or assigning a lock before freeing it, as the compiler may not find every occurrence of such errors.

BEGIN_ORDER, END_ORDER

The BEGIN_ORDER directive identifies an ordered critical region. An ordered critical region begins with a BEGIN_ORDER directive and ends with an END_ORDER directive.

An ordered critical region is a section of code in which execution of one instance of the section must be delayed until previous instances have been executed. In addition, loop iterations must be performed in their correct sequence.

The formats of these directives are as follows:

```
BEGIN_ORDER (lockname)
END_ORDER
```

where *lockname* is the name of a locking variable that was previously assigned by the ASSIGN_LOCK directive.

Example:

```
C$DIR ASSIGN_LOCK (ILOCK)
C$DIR FORCE_PARALLEL
      DO I = 1,N
      A(I) = B(I) * C(I)
C$DIR BEGIN_ORDER (ILOCK)
      IF ( A(I) .LT. 0 ) THEN
         D(I) = D(I-1) + E(I)
      ENDIF
C$DIR END_ORDER
      ENDDO
C$DIR FREE_LOCK (ILOCK)
```

The BEGIN_ORDER and END_ORDER directives must be in the same program unit as the ASSIGN_LOCK and FREE_LOCK directives.

BEGIN_SECTION, END_SECTION

The BEGIN_SECTION directive identifies an unordered critical region. An unordered critical region begins with a BEGIN_SECTION directive and ends with an END_SECTION directive.

An unordered critical region is a section of code in which execution of only one instance of the section is allowed at any one time. Loop iterations need not be performed in sequence.

The formats of these directives are as follows:

```
BEGIN_SECTION (lockname)
END_SECTION
```

where *lockname* is the name of a locking variable that was previously assigned by the ASSIGN_LOCK directive.

Example:

```
C$DIR ASSIGN_LOCK (JLOCK)
C$DIR FORCE_PARALLEL
      DO I = 1,N
      A(I) = B(I) * C(I)
C$DIR BEGIN_SECTION (JLOCK)
      IF ( A(I) .LT. 0 ) THEN
         K = K + 1
      ENDIF
C$DIR END_SECTION
      ENDDO
C$DIR FREE_LOCK (JLOCK)
```

The BEGIN_SECTION and END_SECTION directives must be in the same program unit as the ASSIGN_LOCK and FREE_LOCK directives.

BEGIN_TASKS, NEXT_TASK, END_TASKS

The BEGIN_TASKS directive identifies a sequence of tasks for independent, parallel execution. A sequence of tasks begins with a BEGIN_TASKS directive and ends with an END_TASKS directive. A NEXT_TASK directive precedes each individual task. A task is defined as a sequence of nonloop code that can be executed in parallel.

The following code illustrates the use of the tasking directives:

```
C$DIR BEGIN_TASKS
  statement
    .
    .
    .
C$DIR NEXT_TASK
  statement
    .
    .
    .
C$DIR NEXT_TASK
  statement
    .
    .
    .
C$DIR END_TASKS
```

The preceding example is equivalent to the following loop:

```
C$DIR FORCE_PARALLEL
      DO 100 I = 1,3
        GOTO (10,20,30),I
 10       statement-1
        GOTO 100
 20       statement-2
        GOTO100
 30       statement-3
100   CONTINUE
```

Up to 255 tasks can be specified between a BEGIN_TASKS and an END_TASKS directive.

FORCE_PARALLEL_EXT

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 if the -O3 compiler option is specified. 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. FORCE_PARALLEL_EXT ignores any dependencies between iterations that the compiler locates. When you use this directive on a loop, you may not get correct results; thus, it is important that you check answers generated with 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_PARALLEL

The FORCE_PARALLEL directive tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. Loops can be parallelized with FORCE_PARALLEL whether or not they contain calls.

This directive is effective only if the -O3 compiler option is specified.

FORCE_PARALLEL does not allow interchange or distribution of outer loops for vectorization. FORCE_PARALLEL ignores any dependencies between iterations that the compiler locates. When you use this directive on a loop, you may not get correct results; thus, it is important that you check answers generated with the parallelized code.

If you use this directive with the SCALAR or NO_RECURRENCE directive, an error occurs. In addition, an error occurs when you use FORCE_PARALLEL and another parallelizing directive in the same loop nest.

FORCE_VECTOR

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 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.

The FORCE_VECTOR directive ignores any dependencies between iterations that the compiler locates. When you use this directive on a loop, you may not get correct results; thus, it is important that you check answers generated with the vectorized code.

This directive cannot be used with the SCALAR directive or with the NO_RECURRENCE directive or an error condition results. In addition, an error occurs when you attempt to use FORCE_VECTOR and another vectorizing directive in the same loop nest.

MAX_TRIPS

The MAX_TRIPS directive instructs the compiler that the following loop is never executed more than the specified number of times. The format of this directive is

```
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.

NO_PARALLEL

The NO_PARALLEL directive tells the compiler not to parallelize the loop that immediately follows; vectorization is not prevented.

If the NO_PARALLEL and NO_VECTOR directives both precede a loop, the result is the same as if the SCALAR directive were used.

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`. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more information.

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`. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more information.

NO_RECURRENCE

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.

You must place this directive immediately before a DO statement or a labeled statement that begins a loop. Comment lines can appear between the directive and the start of the loop.

The NO_RECURRENCE directive does not affect recurrences caused by a nested DO loop. The directive can, however, be used on each loop in a nest to give the vectorizer maximum opportunity for improving the performance of the nest.

When the NO_RECURRENCE directive is used, the compiler breaks the recurrence by arbitrarily removing one or more dependencies of the cycle. In the following example, if J is positive, there is no recurrence:

```
C$DIR NO_RECURRENCE
      DO 10 I + 1,N
10    A(I) = A(I+J)
```

The compiler always processes a NO_RECURRENCE directive when the apparent recurrence involves an array element. The compiler always ignores a NO_RECURRENCE directive when the apparent recurrence involves a scalar. In the latter case, the compiler knows that a recurrence exists.

Note

Incorrect results 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.

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. The format of this directive is:

NO_SIDE_EFFECTS (*func* [, *func*])

The parameter *func* specifies one or more user-defined functions.

This directive allows scalar optimization to remove a function call if it occurs in an expression assigned to a scalar variable that is never used. The function call can be removed 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, provided its arguments do not vary within the loop and the call can be moved out of the loop.

NO_VECTOR

The NO_VECTOR directive tells the compiler not to vectorize the loop that immediately follows; parallelization is not prevented.

If the NO_PARALLEL and NO_VECTOR directives both precede a loop, the result is the same as if the SCALAR directive were used.

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. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more information.

PEEL_ALL

The PEEL_ALL directive allows the compiler to peel the loop immediately following the directive, expanding the code without bound. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more information.

PREFER_PARALLEL_EXT

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 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_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 prevents interchange and distribution of outer loops for vectorization. An error occurs when you try to use PREFER_PARALLEL and another parallelizing directive in the same loop nest.

PREFER_VECTOR

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 trys to interchange the loop to be the innermost loop and vectorize it.

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. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more 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. Refer to Chapter 3, "Vector optimization," of the *CONVEX FORTRAN Optimization Guide* for more information.

PSTRIP

The `PSTRIP` directive tells the compiler that the parallel loop immediately following the directive is to be strip mined using the specified length. The format of this directive is

```
PSTRIP (integer_constant)
```

The value *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 strip-mine length for the loop. 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 51 shows the maximum strip-mine lengths used with the `-O3` and `-ep` options.

Table 51
Maximum parallel strip-mine lengths at `-O3`

Processors (`-ep`)	Default compiler length	`PSTRIP`(*k*) Length
1	1	1
more than 1	max(*n*/(2*ep*),1)	*k*

ROW_WISE

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. Reversing the order of subscripts can assist in vectorization. The format of this directive is:

```
ROW_WISE (array_name [,array_name...] )
```

The following cautions apply to the use of the `ROW_WISE` directive:

- Implicit array I/O, such as READ(5,*)A, is not allowed for arrays that appear in a ROW_WISE directive.
- The array appears reversed when viewed in the debugger.
- If the ROW_WISE directive is applied to a dummy argument, the actual argument must also appear in a ROW_WISE directive within the caller. The compiler cannot detect this situation.

The following example illustrates 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 would be 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.

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.

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.

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.

In the following example, the compiler normally interchanges the I loop with the J loop so that elements of A, B, and C are accessed contiguously. The SCALAR directive ensures that the loop of greater iteration count is retained as the innermost loop.

```
C$DIR  SCALAR
       DO 10 I = 1,N              !(where N = 2)
       DO 10 J = 1,M              !(where M = 1000)
10     A(I,J) = B(I,J) + C(I,J)
```

In the following example, neither iteration count is sufficient to warrant vectorizing the loops.

```
C$DIR   SCALAR
        DO 10 I = 1,N           ! (where N = 2)
C$DIR   SCALAR
        DO 10 J = 1,M           ! (where M = 2)
10      A(I,J) = B(I,J) + C(I,J)
```

SELECT

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. The format of this directive is:

SELECT (*vtrip*, *ptrip*, *pvtrip*)

The parameters *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 that 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$DIR SELECT(10,4,20)
C    Run scalar if actual trip count = 1-4.
C    Run parallel if trip count = 5-10.
C    Run vector if trip count = 11-20.
C    Run parallel-vector if trip count>200.
C$DIR SELECT (0,*,*)
C    Run scalar if loop has no vectorizable code.
C$DIR SELECT (*,*,*)
C    Equivalent to C$DIR SCALAR.
```

SYNCH_PARALLEL

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 if the `-O3` compiler option is specified.

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 synchronization, particularly if all the dependencies are in seldom executed branches. The loop in the following example might run faster in a machine with four processors than if it were partially vectorized and the recurrence placed in a scalar, nonparallel loop.

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
```

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 if the -O2 compiler option is specified.

To be eligible for unrolling, a loop must contain no internal branching and 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. Complete unrolling occurs before vectorization, and partial unrolling after vectorization.

VSTRIP

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. The directive has the following format:

VSTRIP (*integer_constant*)

The value *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, 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 flag is 1. At optmization 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 52 shows the maximum strip-mine lengths used with the `-O3` and `-ep` options.

Table 52
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).

Examples:

Table 53 shows the maximum and actual vector strip lengths when the system includes four processors (-*ep*=4).

Table 53
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)

System limits

D

This appendix lists the maximum sizes for the various elements in a CONVEX FORTRAN program.

The maximum...	Is...
Statement length	13,200 characters
Hollerith length	2000 characters
String length	65,535 characters
Identifier length	42 characters
File name length	200 characters
INCLUDE nesting	127
Number of files in a program	127

Additionally, the total space used by all dimensioned arrays in a program must not exceed the memory space of the system architecture.

ASCII character set

This appendix lists the American Standard Code for Information Interchange (ASCII) with each character equivalent in hexadecimal and octal values. The FORTRAN character set is a subset of the ASCII character set; although not all ASCII characters are FORTRAN characters, all of the FORTRAN characters are included in the ASCII character set.

Table 54
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	.
0F	017	SI	1F	037	US	2F	057	/

Table 54
(continued)

Hex value	Octal value	Char	Hex value	Octal value	Char	Hex value	Octal value	Char
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			
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			

FORTRAN 66 compatibility

F

The CONVEX FORTRAN compiler adheres to the American National Standard FORTRAN 77, X3.9-1978, ISO 1539-1980(E). The default language interpretations are FORTRAN 77. The compiler can, however, compile FORTRAN 66 programs. There are five incompatibilities between American National Standard FORTRAN 77 and FORTRAN 66, X3.9-1966:

- `EXTERNAL` statement
- `DO` loop minimum iteration count
- `OPEN` statement `BLANK` keyword default
- `OPEN` statement `STATUS` keyword default
- `X` format edit descriptor

The first two incompatibilities are interpreted by the compiler; the rest are interpreted by the runtime system. If your program uses the `OPEN` statement and you want FORTRAN 66 interpretation rules at runtime, either include a call to `ioinit` in your main program or include the library `I66` in your link by using the option `-lI66` on the `fc` command line. The `X` format edit descriptor use must be modified.

Compiling FORTRAN 66 programs

To compile a FORTRAN 66 program, you can modify the program using the following procedure, transforming it into a FORTRAN 77 program, or use the `-F66` option.

1. Use `grep` to identify `OPEN` statements in which a `STATUS` keyword is to be added, `EXTERNAL` statements that must be changed to `INTRINSIC` statements, and `FORMAT` statements using the `X` edit descriptor. These changed can then me made manually in an editor or automatically through use of a shell script.

2. Use the -F66 option or OPTIONS statement to select FORTRAN 66 language interpretations. The -F66 option allows for the interpretation of EXTERNAL statements, DO loop minimum iteration counts, and BLANK and STATUS keyword defaults in OPEN. It does not affect the X format edit descriptor. If you are running the C shell, you can avoid including the -F66 option in the fc command each time you wish to invoke the FORTRAN 66 compiler by using an alias. alias is a C shell command with the following form:

```
alias fc fc -F66
```

You can include this format in your .cshrc file, with the $ parameter representing specified files. For more information, see the csh(1) man page.

EXTERNAL statement

In FORTRAN 66, the EXTERNAL statement specifies that a symbolic name is the name of either a user-defined external procedure or a FORTRAN-supplied function. In FORTRAN 77, two statements accomplish this function:

- The INTRINSIC statement specifies that the procedure is a FORTRAN-supplied intrinsic procedure, such as SQRT.
- The EXTERNAL statement specifies that the procedure is user-supplied.

Because of the exact specification of these two procedures, you cannot modify the EXTERNAL statements in your program so that the same source program works with FORTRAN 77 and FORTRAN 66. You must substitute an equivalent statement to include the changes, as shown below.

FORTRAN 66	**FORTRAN 77**
EXTERNAL USER	EXTERNAL USER (no change)
EXTERNAL SQRT	INTRINSIC SQRT
EXTERNAL *SQRT	EXTERNAL SQRT (where SQRT is a user function, not the intrinsic for the square root)

DO loop minimum iteration count

In FORTRAN 66 the body of a DO loop is always executed; in FORTRAN 77 the body of the DO loop is not executed if the end condition of the loop is already satisfied when the DO statement is executed. To run a FORTRAN 66 program with the

FORTRAN 77 compiler, you can either use the -F66 option, or modify the DO statements in the program to ensure a minimum loop count of 1.

For example, in FORTRAN 77, the loop

```
DO 20 J=INIT,LAST
```

is not executed if INIT is greater than LAST, but is executed once in FORTRAN 66.

If this DO statement occurs in a FORTRAN 66 program, its equivalent FORTRAN 77 statement is

```
DO 20 J=INIT,MAX(INIT,LAST)
```

OPEN statement keywords

While FORTRAN 66 does not contain an OPEN statement, it does allow for many implementations based on FORTRAN 66 which contain an OPEN statement. Both the BLANK and STATUS keywords in OPEN for FORTRAN 77 differ from the implementations that are used under FORTRAN 66.

BLANK keyword

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D, E, F, G, I, O, and Z field descriptors. In FORTRAN 77, unless the -vfc flag is specified or the COVUE shell is used, the OPEN statement BLANK keyword defaults to BLANK='NULL' (which means that blanks in numeric fields are ignored). The FORTRAN 66 interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

When a logical unit is opened without an explicit OPEN statement and the -F66 option is specified on the compiler command line, CONVEX FORTRAN provides a default that is equivalent to BLANK='ZERO'.

The use of BLANK='NULL' causes embedded and trailing blanks to be ignored and the value converted as if the nonblank characters were right justified in the field. However, the use of BLANK='ZERO' causes embedded and trailing blanks to be treated as zeros.

If your program treats blanks in numeric input fields as zeros, and you do not want to use -1I66 or ioinit, include BLANK='ZERO' in the OPEN statement.

STATUS keyword

The OPEN statement STATUS keyword in FORTRAN 77 specifies the initial status of the file (OLD, NEW, SCRATCH, or UNKNOWN); its default value is UNKNOWN. In FORTRAN 66, where STATUS is called TYPE, the default value is NEW.

If your program assumes that the default value for TYPE is NEW and you do not want to use -1I66 or ioinit, put STATUS = 'NEW' in the OPEN statement.

x descriptor

The FORTRAN 66 implementation of the X format edit descriptor writes blanks to the output record and can extend it. The FORTRAN 77 version does not modify character positions that are skipped and does not, as a result, affect the length of the output record.

Format code separators

Formats without format code separators are supported.

Cray FORTRAN compatibility

To facilitate porting code written for the Cray FORTRANcompiler, the -cfc option can be specified on the fc command line. This Appendix explains the effects this option has on the compiler.

Compiler defaults

The -cfc option changes compiler defaults in the following ways:

- Default data types are as follows:

Type	Default Length
Integer	*INTEGER*8*
Real	*REAL*8*
Complex	*COMPLEX*16*
Logical	*LOGICAL*8*
Double precision	*REAL*16*

- Constants are stored in the default INTEGER or REAL type.
- Constants written in single-precision exponential form (such as 1.23E4) are stored in REAL*8 format.
- Intrinsics that work with default integer and single-precision types work with Cray default types (8-byte quantities).
- *LOGICAL*2* and *4, INTEGER*4, and REAL*4 are treated as *LOGICAL*8*, *INTEGER*8*, and *REAL*8*.
- Double-precision (*REAL*16*) data types are supported. Constants written with a D in the exponent or objects declared double-precision are treated as 16-byte objects.
- Logical constants .T. and .F. are supported.

Note

The CONVEX/VAX representation of REAL*16 data gives more precision (113 bits) than the Cray, IBM, and IEEE specifications. This greater accuracy makes the CONVEX REAL*16 software much slower (possibly 40 times slower) than Cray double-precision software because of the greater number of intermediate results that must be generated and saved internally.

Unsupported Cray features

Only those Cray-specific intrinsics enumerated in the "Supported Cray intrinsics" section of this Appendix are supported. Cray-specific directives are not supported.

In addition, when a program uses the LOC function, Cray FORTRAN returns a word address; CONVEX FORTRAN returns a byte address.

Cray POINTER support

The Cray POINTER statement is supported. However, pointer arithmetic computes byte addresses rather than word addresses. Due to this difference you may need to multiply your pointer offsets by eight. If you are using the pointer in an arithmetic context and if it is declared as a pointer in that subprogram, the compiler flags statements that must be altered.

The syntax for the statement is

```
POINTER(p,s)
```

where *p* is a pointer, and *s* is the name of a local variable or array. In Cray terminology, *s* is the pointee. *s* cannot be associated with any other known piece of named and referenced storage except through assignments to *p* or by associating two or more pointees with one pointer.

No two pointers can point to the same storage, but this optimization requirement is not enforced by the compiler.

Pointees are assumed *not* to overlap. This means that a value stored indirectly through one pointee must not be accessed via another pointee even when both pointees are associated with the same pointer.

Debugging code containing Cray pointers

csd, the CONVEX Symbolic Debugger, is an optional product available as part of the CONVEX Consultant package. It cannot access Cray pointers. However, you can dump the contents of the pointer in csd; this gives you the raw memory address of the pointee. You can then view the contents of this address in csd.

Example:

```
PROGRAM POINT
REAL A(10)
POINTER (IPA, A)
CALL HPALLOC (IPA, 80, IA, IR)
DO I = 1, 10
  A(I) = I
END DO
PRINT *, A
END
```

To examine the contents of A from within csd, first get its address (the value of IPA):

```
(csd) p ipa
2148233220
```

Now you can examine the memory locations beginning at 2148233220 to see the contents of the array A:

```
(csd) 2148233220,10?F
800b9004:  1.000000 2.000000
800b9014:  3.000000 4.000000
800b9024:  5.000000 6.000000
800b9034:  7.000000 8.000000
800b9044:  9.000000 10.000000
```

Refer to the *CONVEX Consultant User's Guide* for more information on the use of csd.

Cray automatic arrays

Cray automatic arrays are supported, under both the -cfc and -f90 compiler options. Any array declared local to a subroutine with one or more non-constant dimensions is considered an automatic array. Memory for automatic arrays is dynamically allocated on the stack on entry into the subroutine based on a

variable dimension value passed into the subroutine. This stack space is freed on exit from the subroutine; automatic arrays cannot be saved using the SAVE statement.

Automatic arrays are described in detail in the "Automatic arrays" section of Chapter 2, "FORTRAN statement components."

Cray BUFFERIN, BUFFEROUT support

CONVEX provides a look-alike implementation of the Cray BUFFERIN, BUFFEROUT feature and its accompanying routines. This feature is provided mainly to aid in porting existing Cray code; it is not likely to provide noticeably superior performance compared to conventional Convex I/O methods. BUFFEROUT will always be slightly slower than unformatted fixed record length I/O.

Related statements and routines

The following statements are available for use with BUFFERIN, BUFFEROUT:

```
BUFFERIN (unit, mode) (beginLoc, endLoc)
BUFFEROUT (unit, mode) (beginLoc, endLoc)
```

These library routines are also available: UNIT(*unit*), LENGTH(*unit*), GETPOS(*unit*) and SETPOS(*unit*, *pos*). These routines reside in `libcfc.a`, which is implicitly loaded when linking with the `-cfc` option; if you do not compile using `-cfc` these routines will not be available. Old object files will link and run with the new libraries.

Note

If present, data format conversions specified in the OPEN statement do not affect data read or written with BUFFERIN or BUFFEROUT statements.

Restrictions

Note the following restrictions:

- The SETPOS function cannot be used with magnetic tape; doing so produces an error message.
- Using BUFFERIN and BUFFEROUT with data types less than eight bytes flags a fatal compiler error.
- Other FORTRAN I/O statements (READ, WRITE, PRINT, ACCEPT, TYPE) cannot be used on the same unit as BUFFERIN, BUFFEROUT.
- BACKSPACE does not work with BUFFERIN, BUFFEROUT files.

Cray unformatted file support

CONVEX FORTRAN is capable of reading and writing unformatted Cray data files through use of the FORMAT = UNFORMATTED/CRAY and FORMAT = UNFORMATTED/CRAYUB keyword definitions in the OPEN statement. Formatted files are supported regardless of file structure or access mode; unformatted files are partially supported depending on file structure and access mode. Table 55 shows the degree of support provided given various combinations of file structures and access modes.

Table 55
Unformatted Cray files readable by CONVEX FORTRAN

Cray file structure	Acces mode	
	Sequential	**Direct**
Unblocked (pure)	Does not comply with ANSI FORTRAN 77 standard; no record boundaries inherent in file; BACKSPACE not permitted; use FORMAT=UNFORMATTED/CRAYUB in OPEN statement.	CONVEX FORTRAN reads directly.*
Blocked	Must run fcUnblock† utility, supplied with CONVEX FORTRAN V7.0, on these files to convert them into a form readable by CONVEX FORTRAN. Use FORMAT=UNFORMATTED/CRAY.*	Cray does not allow these.

*These are the default file structures written in Cray FORTRAN.
†For more information on the fcUnblock utility, see the fcUnblock(1F) man page.

Supported Cray library routines

The following library routines are available using the `-cfc` option:

cft$bool	cft$dprod	clock	date	dump
etime	findch	gather	hpalloc	hpcheck
hpclmove	hpdeallc	hpdump	hpnewlen	hpshrink
iceil	igtbyt	ihplen	ihpstat	ilsum
int24	jdate	komstr	lint	mvc
pack	putbyt	rbn	rnb	scopy
sdot	second	ssum	strmov	timef
tr	unpack			

Additionally, most `libU77.a` routines can be accessed in Cray mode. The exceptions are: `dbesj0`, `dbesj1`, `dbesjn`, `dbesy0`, `dbesy1`, `dbesyn`, `derf`, and `derfc`. For more information about `libU77.a` routines, refer to the intro.3f man page.

Note

A Cray mode call to `date` accesses the Cray specific date routine, which returns the date in a format different from that returned from the `libU77.a` `date` routine. Similarly, a Cray-mode call to `rand` accesses the Cray routine and returns a different sequence of numbers than the corresponding call to the `libU77.a` `rand` routine.

Supported Cray intrinsics

The following intrinsics are available using the `-cfc` option:

and	compl	cot	csmg	cvmgm
cvmgn	cvmgp	cvmgt	cvmgz	dcot
eqv	leadz	loc	mask	movbit
neqv	numarg	or	popcnt	poparr
ranf	ranget	ranset	shift	shiftl
shiftr	xor			

Additionally, all Military Standard (MIL-STD-1753) routines are supported from Cray mode.

Cray TASK COMMON support

CONVEX FORTRAN supports Cray TASK COMMON blocks under the `-cfc` option. These blocks are created using the TASK COMMON statement, which has the following form:

```
TASK COMMON /cbn/nlist[,/cbn/nlist]...
```

where

cbn
: is a symbolic name for a task common block. Unnamed TASK COMMON blocks are not allowed.

nlist
: is a list of variable names, array names, and array declarators. These variables cannot appear in a DATA statement, but otherwise can be used like any variables in COMMON storage.

All occurrences of the TASK COMMON block must be declared TASK COMMON; a common block cannot be declared both COMMON and TASK COMMON. TASK COMMON blocks can only be declared in functions, subprograms and BLOCK DATA subprograms. Variables in TASK COMMON blocks are provided thread-local storage. A program should already be running multiple threads before calling a subroutine that contains a TASK COMMON block.

Cray Boolean octal constant support

CONVEX FOTRAN supports Cray Boolean octal constants under the `-cfc` option. Cray Boolean octal constants consist of one or more octal digits followed by the letter B. A boolean octal constant has the following form:

*cc...c*B

where *c* represents an octal digit. There can be up to 22 octal digits in an octal constant. An octal digit can range from 0 to 7.

Like all Cray constants, octal constants occupy 8 bytes (equivalent to one 64-bit Cray word) in memory. It follows that octal constants can be 22 octal digits long, but, because 22 octal digits represent 66 bits in memory, the leftmost octal digit must be either 0 or 1. Octal constants are right justified and zero-filled to the left. An octal constant that occupies all 22 digits cannot have a value greater than 1 as its leftmost octal digit. Blanks within an octal constant are ignored.

Examples:

Valid	Invalid	Reason
`765B`	`835B`	8 not in range 0 to 7
`1126752354176524376524B`	`3126752354176524376524B`	Leftmost digit 1

Cray Hollerith constants

CONVEX FORTRAN supports left justified Cray Hollerith constants of the following form:

*n*L*cc...c*

or

'*cc...c*'L

where

n
: specifies the number of the characters in the constant (including spaces and tabs). The value of *n* must be an unsigned positive integer greater than zero.

c
: is a printable ASCII character.

Cray hollerith constants occupy 8 bytes (equivalent to one 64-bit Cray word) in memory. Each byte contains one ASCII character code, and hollerith constant using this form cannot exceed eight characters in length. This form left justifies the constant in memory and zero-fills its 8-byte storage space to the right. You must supply the the `-cfc` option when using this form.

Example:

Valid	Invalid	Reason
`4LHelp`	`4L'Help'`	No quote in this form.
`'foo'L`	`3'foo'L`	Number of characters specifier not allowed in this form.

VAX FORTRAN compatibility

H

This appendix describes compatibility between VAX FORTRAN and CONVEX FORTRAN. To facilitate porting code written for the VAX FORTRAN compiler, certain VAX features are supported as described below.

Supported features

CONVEX FORTRAN supports the following VAX FORTRAN features only when the `-vfc` option is specified on the `fc` command line:

- VAX `INCLUDE` statement
- `REAL*16` data type
- The alternate form (without parentheses) of the `PARAMETER` statement with only one constant specified
- The `'r` form of the record specifier
- Octal constants in the form "*nn*, where *nn* is a string of octal digits
- Default file names in the form `FOR0`*nn*`.DAT`, where the number *nn* corresponds to unit number *nn*
- VAX FORTRAN records
- Hollerith constants where a `CHARACTER` value is expected
- The `RECL=` specifier used in the `OPEN` and `INQUIRE` statements. This specifier returns the number of VAX words rather than bytes for unformatted files. (This is not true for programs compiled and loaded separately unless `-vfc` is specified for the load phase.)

The organization and structure of VAX FORTRAN records is discussed briefly at the end of this appendix.

Unsupported features

CONVEX FORTRAN does not support the following VAX FORTRAN features:

- %DESCR
- RMS calls
- The VOLATILE statement
- The zccc...c form of hexadecimal constants
- Interactive display of NAMELIST group and values or end-of-line comments (!) in the NAMELIST input data
- Extra parentheses in WRITE statements (allowed in VAX FORTRAN)
- VMS file names
- Byte ordering with respect to passing characters and parameters
- Logical values. On CONVEX computers, a logical value is true if it is all 1. On VAX, it is true in a test if the low-order bit is 1, for example, IF(A) is equivalent to IF(A .AND. 1).
- Numerical differences. The accuracy of CONVEX floating-point representation and the rounding method used cause these differences. Refer to the *CONVEX Architecture Reference* for further information.
- Automatic conversion of REAL*8 (in caller) to REAL*4 (in called subroutine) across subroutine calls
- Calling a function as a subroutine
- Radix 50 constants
- The variable on the left side of a character assignment statement appearing on the right side (VAX FORTRAN extension)
- MOD function defined for a zero denominator
- Modifying an argument within a subroutine if the subroutine was called with a constant in the argument list (the CONVEX FORTRAN compiler enforces this rule, which is an ANSI standard)
- DO statements of the form:

  ```
  DO 714 J=1, 100 WHILE (Q.NE.Z)
  ```

- VMS pathnames. The FILE name you specify when using the INQUIRE statement under the COVUE shell must be the absolute UNIX pathname.

- The FILE keyword in the OPEN statement. The keyword must specify the name of the file to open as a character expression; numeric variables, arrays or array elements containing the file name cannot be substituted.

CONVEX FORTRAN does not support these VAX FORTRAN I/O extensions:

- REWRITE, DELETE, and UNLOCK statements
- Indexed I/O (key-indexed files)
- File sharing
- DEFINEFILE statement
- OPEN keywords (PRINT and SUBMIT values for DISPOSE; USEROPEN; INITIALSIZE; EXTENDSIZE; BUFFERCOUNT; SEGMENTED for RECORDTYPE; and ORGANIZATION)
- CLOSE keywords (PRINT and SUBMIT values for STATUS)
- ASCII null as a carriage-control character

The internal format of variable-length type records of VAX FORTRAN and CONVEX FORTRAN differ when RECORDTYPE=VARIABLE.

VAX and CONVEX versions of the STOP message differ as follows:

Statement	CONVEX message	VAX message
STOP	STOP:	FORTRAN STOP
STOP 4	STOP: 4	4
STOP 'here'	STOP: here	here

Integer overflow traps are turned off by default in fc, whereas VAX FORTRAN enables them by default. The main reasons for this are:

- Overflow is turned off in C, and many users mix C and FORTRAN code.
- It is difficult to optimize integer expressions if integer overflow is turned on because most addresses are negative integers near overflow on a CONVEX machine.

When you use the H descriptor on input, the first character transferred appears immediately after the letter H. The characters that are in the field descriptor before input are replaced (overwritten) by the new input characters.

Miscellaneous differences

The following miscellaneous differences exist between VAX FORTRAN and CONVEX FORTRAN:

- Unit numbers in VAX FORTRAN range from 0 to 99 and in CONVEX FORTRAN from 0 to 255.
- VAX FORTRAN supports the use of Hollerith and apostrophe edit descriptors during formatted input. CONVEX FORTRAN allows Hollerith descriptors but does not allow apostrophe edit descriptors.
- In VAX FORTRAN, `'cc...c'O` octal constants are of type `INTEGER`; in CONVEX FORTRAN, the `'cc...c'` form of octal constants is typeless.
- In the `ASSIGN` statement, `ASSIGN` *s* `TO` *i*, where *s* is the label of an executable statement in the current program unit and *i* is an integer variable, VAX FORTRAN sets the integer variable to 1 at initial execution of the `ASSIGN` statement; CONVEX FORTRAN sets it to 0.
- If invalid data is encountered on a VAX FORTRAN `READ` statement, all variables on the `iolist` are assigned except those corresponding to the bad data. If the same error occurs in CONVEX FORTRAN, the `READ` statement ends at once and the remaining variables in the *iolist* are unchanged.
- VAX FORTRAN correctly handles *`REAL*16`* constants that do not contain `Q` in the exponent. CONVEX FORTRAN does not; if `Q` is not specified, the constant is considered to be `REAL*4`.
- CONVEX FORTRAN follows the ANSI standard in that any function referenced directly or indirectly in an I/O statement cannot contain another I/O statement. VAX FORTRAN may not always follow this standard.

VAX FORTRAN records

A VAX FORTRAN record contains one or more fields. Fields within a record are defined by a structure declaration that defines field names, types of data within fields, and order and alignment of fields within a record.

Structure declaration

A structure declaration is bounded by STRUCTURE and END STRUCTURE statements and has one or more field declarations. The order in which the field declarations occur determines the order of fields within the structure. At least one field declaration must be specified or an error condition occurs.

A structure declaration has the following format:

```
STRUCTURE /structure-name/
    field declaration
        .
        .
        .
END STRUCTURE
```

A structure declaration does not create a variable. A variable is created by a RECORD statement containing the name of a previously declared structure. The format of a record statement is

```
RECORD /structure-name/record-namelist
```

where *structure-name* is the name of a previously declared structure and *record-namelist* is a list of variable names, array names, or array declarations, separated by commas.

Records must be read and written using unformatted I/O. For example,

```
C DEFINITION OF THE NAME STRUCTURE
       STRUCTURE / NAME /
          CHARACTER*5 LAST
          CHARACTER*5 FIRST
          CHARACTER*1 INITIAL
       END STRUCTURE

C DEFINITION OF THE PERSON STRUCTURE
      STRUCTURE / PERSON /
         RECORD / NAME / NAME
         LOGICAL*1 SEX
         RECORD / DATE / BIRTH
   END STRUCTURE
```

```
C SETUP NUMBER OF EMPLOYEES TO BE HANDLED
      RECORD / PERSON / EMPLOYEES(3)

C READ EMPLOYEE DATA
      DO I = 1,3
         READ(2) EMPLOYEES(1)
      ENDDO
```

Field declaration

A field declaration can be any combination of the following:

- A typed data declaration
- A substructure declaration
- A union declaration.

A typed data declaration is the same as a normal FORTRAN type statement. As with FORTRAN typed data statements, field declarations can contain initializers. The name `%FILL` can be used in place of a field name to create space in the structure for padding. This space cannot be initialized.

A substructure must be declared by a *RECORD* statement that creates an instance of a previously declared structure, not by a nested `STRUCTURE` statement. The preceding example shows the proper declaration of a substructure within the `PERSON` structure.

A union declaration is bounded by `UNION` and `END UNION` statements and defines a data area that can be shared during program execution. A union declaration must contain at least two map declarations (as indicated below) or an error condition occurs. A union declaration has the following format:

```
UNION
   map declaration
   map declaration
     .
     .
     .
END UNION
```

A *map-declaration* defines a unique group of fields and is bounded by MAP and END MAP statements. A map declaration must contain at least one field declaration or an error condition occurs. A map declaration has the following format:

```
MAP
      field declaration
        .
        .
        .
END MAP
```

Sun FORTRAN compatibility

To facilitate porting code written for the Sun FORTRAN compiler, the `-sfc` option can be specified on the `fc` command line. When used, this option changes certain aspects of the CONVEX FORTRAN compiler as follows:

- As in the C language, escape sequences using a backslash (\) are supported in character strings to define nonprintable characters. The following table lists the supported sequences.

Character	Sequence
newline	`\n`
tab	`\t`
form feed	`\f`
NUL	`\0`
single quote	`\'`
double quote	`\"`
backslash	`\\`

- The ampersand (`&`) character in the first nonblank column of a source line indicates that the line is a continuation, regardless of what is in column 6.
- Recursive subroutines and functions are allowed.
- The declarations `AUTOMATIC` and `STATIC` are supported.

The preceding features, which are implemented by CONVEX FORTRAN, represent a subset of Sun FORTRAN features. The Sun definition of real constants as 64-bit numbers is not supported.

Part 3
CONVEX FORTRAN Man Pages

This part contains hard copy versions of sections 1F and 3F of the online man pages. This hard copy information was formerly contained in the *CONVEX FORTRAN Programmer's Reference.*

Section 1F of the man pages includes those pages that describe the CONVEX FORTRAN compiler (`fc`) and utilities provided with it.

Section 3F of the man pages include those pages that describe the library functions contained in the CONVEX FORTRAN utility library.

intro(1F)

Introduction to FORTRAN utilities

Description

This section describes those commands that are provided with the CONVEX optimizing FORTRAN compiler. These commands assist in developing, porting, and compiling FORTRAN source files into executable programs. They are installed into `/usr/convex`.

Commands

Name	Description
`fc`	CONVEX vectorizing FORTRAN compiler
`fct`	convert a FORTRAN file for `fc`
`fcUnblock`	unblock Cray blocked file format
`fpr`	print FORTRAN file
`fsplit`	split a multi-routine FORTRAN file into individual files
`fcxref`	current FORTRAN cross-reference generator
`fxref`	old FORTRAN cross-reference generator; being phased out

fc(1F)

CONVEX FORTRAN compiler

Synopsis

`fc` [*option*][...] *file* ... [*loader_option*][...]

Description

`fc` is the CONVEX FORTRAN compiler. It accepts several types of arguments:

Arguments beginning with - are options; other arguments are names of files to process.

Filenames with a suffix of `.f` or `.FOR` must be FORTRAN source programs; they are preprocessed (if the `-fpp` option is used) and compiled. Each object program is placed in a file in the current directory with the same root name as the source filename and a suffix of `.o`.

Filenames with a suffix of `.s` must be assembly source programs and are assembled, producing a `.o` file.

If a single source file is compiled and loaded by a `fc` command, the `.o` file is deleted.

Other files can be object files or libraries; they are passed to `ld` to be linked together with the objects from all compilations and assemblies.

Preprocessor

Option	Meaning
`-D` *name*[=*def*]	Defines *name* to the preprocessor, as if it had been specified in a `#define` statement. If no definition is given, *name* is defined as 1. No more than 55 `-D` options can appear on a command line.
`-E`	Runs only the FORTRAN preprocessor on the named FORTRAN programs, and sends the result to the standard output.
`-I` *dir*	For `#include` (or `INCLUDE`) files whose names do not begin with `/`, `fpp` always seeks first in the directory of the file argument, then in directories named in `-I` options, then in directories on a standard list. Directories given

in -I options are searched in the order in which they appear on the command line. No more than 8 directories can be specified.

-U *name*	Removes any initial definition of *name*. The only built-in names defined by fc are __LINE__, and __FILE__.
-fpp	Selects the FORTRAN preprocessor to run as the first step of compilation. If this option is not specified, the preprocessor is not used.
-pp=*name*	Selects a user supplied preprocessor (*name*) to be run after the FORTRAN preprocessor (-fpp). If -fpp is not specified, the user supplied preprocessor is the first step of compilation. The user supplied preprocessor may be a shell script or executable program. It will be passed two arguments, an input file and a output file. The user supplied preprocessor should exit with a zero status if compilation should continue. A non-zero status will terminate further processing of the current input file.

Language Compatibility

Option	Meaning
-F66	Selects FORTRAN-66 compiler interpretation rules in cases of incompatibility. This option can also appear in the OPTIONS statement.
-cfc	Selects Cray FORTRAN compiler (cft77) language definitions instead of the CONVEX FORTRAN definition. This option cannot be used with the -r*n* or -i*n* options.
-f90	Allows use of the Fortran 90 features included in CONVEX FORTRAN Version 7.0. These include some Fortran 90 intrinsic functions (refer to Appendix A of *CONVEX FORTRAN Language Reference Manual*) and Fortran 90 array notation (refer to Chapters 2 and 5 of *CONVEX FORTRAN Language Reference Manual*).

`-sa`	Prevents FORTRAN from generating pre-compiled argument packets in the text segment. Instead, all arguments are placed on the stack. This option should only be used when an application contains C programs called from FORTRAN, as using it with applications coded only in FORTRAN slows down the application. This option can also appear in the OPTIONS statement.
`-sfc`	Accepts certain language extensions implemented in the Sun FORTRAN compiler (f77).
`-vfc`	Accepts certain language extensions implemented in the VAX FORTRAN compiler that either conflict with current or proposed FORTRAN standards, or that cause compatibility problems with existing CONVEX FORTRAN programs.
`-dfc`	Use `-dfc` only with the `-vfc` option and only when you use the Binary Data File Format Conversion feature (see Chapter 7 of the *CONVEX FORTRAN Language Reference Manual*). This option causes all references to VAX records in and I/O statement to be decomposed.

Optimization

The `-O`*n* option performs machine-independent optimizations, level `n`, where:

Value of *n*	Optimization
0	Selects basic block machine-independent scalar optimization;
1	Selects `-O0` plus program unit machine-independent scalar optimization;
2	Selects `-O1` plus vectorization;
3	Selects `-O2` plus parallelization.

If the `-On` option is not used, the compiler performs no machine-independent optimization. This option can also appear in the OPTIONS statement.

Other optimization options comprise the following:

Option	Optimization
`-ds`	Causes the compiler to automatically select loops to replicate and to compile several versions of such loops. The compiler then dynamically selects the version of each loop to be executed. This option is available only at optimization level `-O2` or `-O3`.
`-ep`*n*	Specifies the number of processors that are expected to be available to execute the program. Optimizes single-program performance at the expense of system throughput.
`-il`	Generate inline substitution intermediate files for each routine. No object or assembler files are produced and the loader is not invoked.
`-is` *directory*	Perform inline substitution using the `.fil` files in the given *directory*.
`-no`	Perform no optimization. This option can also appear in the OPTIONS statement. This option is the default if the -O option is not specified.
`-nopeel`	Disallows loop peeling, which is enabled by default. Refer to `-peel` and `-peelall` below. Refer also to the *CONVEX FORTRAN Optimization Guide* for more information.
`-noptst`	Disallows test promotion, which is enabled by default. Refer to `-ptst` and `-ptstall` below. Refer also to the *CONVEX FORTRAN Optimization Guide* for more information.
`-peel`	Causes removal of 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 will peel boundary values and expand code up to a predetermined conservative limit. With the `-peel` option, this limit is increased and code expansion may become significant. `-peel` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-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 to the point of exceeding the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options. Refer to the `CONVEX FORTRAN Optimization Guide` for more information.

`-ptst`
: Causes a test to be promoted out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loop contains fewer tests than the original (or no tests at all), so it executes much faster. By default, the compiler promotes tests and replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticable increase in compile time. `-ptst` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-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. `-ptstall` must be used with the `-O2` or `-O3` optimization options. Refer to the *CONVEX FORTRAN Optimization Guide* for more information.

`-rl`	Performs loop replication optimizations (loop unrolling, dynamic code selection) on loops selected on the basis of profitability. This option can also appear in the OPTIONS statement.
`-uo`	Performs potentially unsafe optimizations, i.e., moves the evaluation of common subexpressions and/or invariant code from within conditionally executed code. Such moved code is executed unconditionally. This option can also appear in the OPTIONS statement.
`-ur`	Causes the compiler to automatically select and unroll loops. This option is available only at optimization level `-O2` or `-O3`.

Code Generation

Option	Meaning
`-S`	Generates symbolic assembly code for each program unit in a source file. Assembler output for a source file `x.f` is put on file `x.s`; the assembly file is not assembled.
`-c`	Suppresses the loading phase of the compilation. Output from the files `x.f` or `x.s` is put on `x.o`.
`-fi`	Specifies that real constants are to be translated into IEEE format. This option requires that the machine be equipped with the IEEE support hardware. CONVEX IEEE support hardware does not support the full IEEE 754 standard. If no floating point format is specified, the site default is used.
`-fn`	Specifies that real constants are to be translated into native CONVEX format and processed in native mode. If no floating point format is specified, the site default is used.
`-i`*n*	Specifies that `INTEGER` and `LOGICAL` variables declared with unspecified length will occupy *n* bytes of storage. Transforms intrinsic function references that return default integer or logical

values to return integer values of the specified length. Note that storage association between integer and real data will not conform to the ANSI FORTRAN-77 Standard if the size of an integer datum is not the same as the size of a real datum. *n* can be 2, 4, or 8; the default value is 4. This option can also appear in the OPTIONS statement.

`-p8`

Specifies that default `INTEGER`, `LOGICAL`, and `REAL` values will occupy 8 bytes of storage, and `DOUBLE PRECISION` and `COMPLEX` values will occupy 16 bytes; transforms intrinsic function references that accept or return default `INTEGER`, `LOGICAL`, `REAL`, `COMPLEX`, or `DOUBLE PRECISION` values to accept values of the specified size. Storage association and intrinsic function references still fully comply with the ANSI FORTRAN-77 standard. The user can override the default size of any variable with length override specifications in its declaration, and can use intrinsic functions that are defined to accept or return `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL` values of specific size.

`-pd8`

Specifies that default `INTEGER`, `LOGICAL`, `REAL`, and `DOUBLE PRECISION` values will occupy 8 bytes of storage, and `COMPLEX` values will occupy 16 bytes; transforms intrinsic function references that accept or return default `INTEGER`, `LOGICAL`, `REAL`, `COMPLEX`, or `DOUBLE PRECISION` values to accept values of the specified size. Intrinsic function references still comply fully with the ANSI FORTRAN-77 standard; storage association with `DOUBLE PRECISION` data and data of any other type will not conform to the ANSI FORTRAN-77 Standard. The user can override the default size of any variable with length override specifications in its declaration, and can use intrinsic functions that are defined to accept or return `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL` values of specific size. This option allows programs to take advantage of increased `REAL` precision without paying the speed penalty of `REAL*16` arithmetic.

`-r`*n*	Specifies the number of bytes occupied by `REAL` variables declared without length specification. For compatibility with previous releases, does not affect the meaning of intrinsic function references. Note that storage association between real or complex data and double precision data will not be conformant to the ANSI FORTRAN-77 standard. *n* can be 4 or 8; the default value is 4. This option can also appear in the OPTIONS statement. When possible, use the `-p8` and `-pd8` options instead.
`-re`	Specifies that the compiler generate re-entrant code by placing local variables in automatic storage. This option should be used to compile subroutines that will be called recursively or from parallel loops.
`-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.
`-tm` *x*	Specifies the target machine architecture. *x* can be c1 (or, equivalently, C1), c2 (or C2), c32 (or C32), c34 (or C34) or c38 (or C38). When this option does not appear the target machine architecture is the type of machine the compiler is running on. Note that, due to subtle differences between executables on different machines, the `file` utility may not recognize a C2 or C3 series executable, even if the `-tm` option was used at compile time.

Debugging, Profiling

Option	Meaning
`-a1`	Causes non-character arrays declared with a last dimension of 1 to be treated as if they were declared as assumed-size arrays (with a last dimension of *). The effect is to make subscript checking work if `-cs` is also used. This option can also appear in the OPTIONS statement.

`-cs`
: Compiles code that checks that each subscript is within its array bounds. Does not check bounds for arrays that are dummy arguments for which the last dimension bound is specified as *. This option can also appear in the OPTIONS statement.

`-cxdb`
: Produces additional information that allows you to use CXdb, the CONVEX Visual Debugger, to debug your program. CXdb is a symbolic debugger with a graphical user interface that uses multiple windows and allows you to debug optimized code. `-cxdb` cannot be used if the `-S` or `-pb` command line options are used. When this option is included on the command line, a directory called `.CXdb` is created in your current working directory that contains auxiliary information. Refer to *CONVEX CXdb User's Guide* for more information. CXdb is an optional product.

`-db` or `-g`
: Produces additional information for use by the symbolic debugger, `csd` and passes the `-lg` option to `ld`. This option can be used with all levels of optimization. If the `-O`*n* option is used to specify optimization at some level, there can be source statements for which no debugging information is generated.

`-dc`
: Specifies that a line with a `D` in column 1 is to be compiled (not treated as a comment line).

`-p`
: Generates code that counts the number of times each routine is called. If loading takes place, this option replaces the standard startup routine with one that automatically calls `monitor`(3) at the start and arranges to write out a `mon.out` file on normal termination of the program. Causes `fc` to tell the loader to search profiling libraries instead of the standard libraries. An execution profile can then be generated with `prof`(1) (optional product).

`-pa`
: Produces information in the object file for use with the CXpa routine and loop profiling features. See the CXpa documentation for a full description. (CXpa is an optional product.)

`-pab`	Produces information in the object file for use with the CXpa block profiling feature. See the CXpa documentation for a full description. (CXpa is an optional product.)
`-par`	Produces information in the object file for use with the CXpa routine profiling feature. See the CXpa documentation for a full description. (CXpa is an optional product.)
`-pb`	Produces source-level counting code that produces an execution profile named `bmon.out` on normal termination. Listings of source-level execution counts can then be obtained with `bprof`(1) (optional product).
`-pg`	Produces counting code in the manner of `-p` but invokes a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file on normal termination. An execution profile can then be generated with `gprof`(1) (optional product).
`-sc`	Stops compilation of each program unit after the program has been determined to be a valid FORTRAN program. No assembler, object, or executable file is generated.

Compiler Output

Option	Meaning
`-LST`	Produces a source listing on `stdout`. `include` files are not listed. Errors and informational messages are inserted in the listing at appropriate points. The output is paginated suitably for most hardcopy printers.
`-LSTI`	Similar to `-LST`, except that `include` files are listed also.
`-na`	Suppresses all advisory diagnostic messages. This option can also appear in the OPTIONS statement.

`-link` *-arg*	Passes *arg* to the loader where *arg* is an arbitrary loader option. You must delimit *arg* with quotation marks if a blank space appears in the argument. Note that a library linked with the `fc` driver cannot contain a C main program.
`-nv`	This option is no longer supported. Use `-or none` to suppress vectorization summary messages.
`-nw`	Suppresses all warning diagnostic messages. This option can also appear in the OPTIONS statement.
`-or` *table*	Controls the contents of the optimization report. *table* can be `all`, `none`, `loop`, or `array`. If `-or` is not specified, then only the loop table is produced by the optimizer.
`-xr`	Invokes the cross-referencer and generates a cross-reference report at compilation time.
`-xra`	Invokes cross-referencer and generates data file but does not generate report. The report can be generated by running `fcxref`. Refer to Chapter 4 for more information on `fcxref`.
`-xro`	Invokes the old cross referencer, `fxref`. See the `fxref`(1F) man page for more information.
`-xrl`	Calls the old cross referencer, `fxref` and puts all objects (variables, arrays, etc.) into one big table, rather than print a separate table for each class of objects.

The following options can be used with the `-xr` and `-xra` options:

Option	Meaning
`-iw` *n*	Controls the column width for identifiers. Default value is *n* = 16.
`-pl` *n*	Specifies a default maximum page length of *n* lines per page. Default is *n* = 60.

-pw *n*	Specifies a desired page width of *n* columns per line. Data that exceeds this width is wrapped to the next line. Default is *n* = 80.
`-xrf` *file*	Use *file* as the cross-referencer data file instead of `.fcxrefData`.
`-xrr` *file*	Sends the cross-reference report to *file* rather than to `stdout`.
`-xrg` *n*	Produces composite/global common block reports after all routine reports. Determine *n* by summing the desired report formats from the list shown in the `fcxref`(1F) man page.
`-xro`	Invokes the old cross referencer, `fxref`. See the `fxref`(1F) man page for more information.
`-xrl`	Calls the old cross referencer, `fxref` and puts all objects (variables, arrays, etc.) into one big table, rather than print a separate table for each class of objects.

Miscellaneous

Option	Meaning
`-ansic`	Links `/usr/lib/libc.a` to your program. `libc.a` is typically 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 allow you to override a `-pcc` (see below) specified in FCOPTIONS. In addition to the different library links, using `-ansic` instead of `-pcc` will produce signal #6 instead of signal #4 on multiple library aborts and print IOT instead of "Illegal Instruction" when they abort.

`-B` *dir*	Finds the substitute compiler (`fskel`, `fpp`, and `errmsgf`) in the directory *dir*. *dir* is optional; the default directory is `/usr/convex/oldfc`, where the `fc` installation process saves a backup version of the compiler. Use as follows:

`/usr/convex/oldfc/fc -B/usr/convex/oldfc` [*options* ...] *file* ...

`-nosc`	Disables short circuiting of conditional expressions. Refer to Chapter 6 of the *CONVEX FORTRAN Language Reference Manual* for more information.
`-o` *name*	Specifies that *name* is the name of the executable file produced by `ld`. If this option is not specified, the default name is `a.out`. If `ld` is not invoked because `-c` was specified and there is only one file to compile or assemble, *name* is the name of the object module produced.
`-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.
`-tl` *time*	Specifies the maximum CPU time (in minutes) to be used in this compilation. If the allotted time is exceeded, the compilation aborts.
`-vn`	Identifies compiler version. Output goes to `stderr`.
`-72`	The compiler normally processes all characters of each source program line. The `-72` option forces the compiler to process only the first 72 characters of each source program line. A line with fewer than 72 characters is treated as if it were padded with blanks to be 72 characters long; in the default mode no padding occurs. A tab character is counted as only one character, so a file using tabs cannot compile as expected. `fct`(1F) or `expand`(1) can be used to expand tabs into an equivalent number of spaces.

Loader Options

These options are passed to `ld(1)`. See the *CONVEX Compiler Utilities User's Guide* for their meaning and use:

`-D -F -L -M -T -X -d -e -l -m -r -s -t -x -y`

Environment

The environment variable `FCOPTIONS` can be preset so that options need not be specified every time `fc` is invoked. The value of the `FCOPTIONS` environment variable is `prepended` to the command line. For example,

```
% setenv FCOPTIONS '-xr -sl'
```

will cause a cross reference table and source listing to be generated for every compilation.

Files

file.[f,FOR,s]	input file
file.o	object file
a.out	generated executable program
/usr/convex/fskel	compiler
/usr/convex/fpp	FORTRAN preprocessor
/usr/convex/errmsgf	text of FORTRAN compiler error messages
/usr/convex/fxprint	formatter for FORTRAN cross-reference table
/usr/convex/fxnames	generator for FORTRAN cross-reference table
/usr/lib/libF77.a	intrinsic function library
/usr/lib/libF77_p.a	profiled intrinsic function library
/usr/lib/libI77.a	FORTRAN I/O library
/usr/lib/libI77_p.a	profiled FORTRAN I/O library
/usr/lib/libU77.a	interface library
/usr/lib/libU77_p.a	profiled interface library
/usr/lib/libU77pd8.a	interface library for `-cfc` and `-pd8` modes
/usr/lib/libU77p8.a	interface library for `-cfc` and `-p8` modes
/usr/lib/libcfc.a	Cray mode support 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-unix filename translation routines
/usr/lib/libD77.a	dummy VMS-to-unix filename translation routines
/usr/lib/libc.a	C library: see section 3
/usr/lib/libc_old.a	C library for `-pcc` mode: see section 3
/usr/lib/libc_p.a	profiled C library
/usr/lib/libC1.a	math library for C1s (extended ANSI C version)
/usr/lib/libC1_old.a	math library for C1s (-pcc version)
/usr/lib/libC2.a	math library for C2s (extended ANSI C version)
/usr/lib/libC2_old.a	math library for C2s(-pcc version)
/usr/lib/libmathC1.a	Math library optimized for the C1
/usr/lib/libmathC1_p.a	Profiled math library optimized for the C1
/usr/lib/libmathC2.a	Math library optimized for the C2

/usr/lib/libmathC2_p.a	Profiled math library optimized for the C2
/usr/lib/*_pa.a	many of these libs are profiled for use with CXpa
mon.out	file produced for analysis by `prof`(1)
bmon.out	file produced for analysis by `bprof`(1)
gmon.out	file produced for analysis by `gprof`(1)

See Also

`prof(1), bprof(1), gprof(1), csd(1), ld(1), fct(1F), fpr(1F), fsplit(1F), pmd(1)`
CONVEX FORTRAN Language Reference Manual
CONVEX FORTRAN User's Guide

Diagnostics

The diagnostics produced by `fc` itself are self-explanatory. Occasional messages can be produced by the loader.

Notes

`fc` is an optional product; for more information, contact your CONVEX sales representative.

Warnings

The Korn Shell (`ksh`) contains an in-line editor which is invoked using the `fc` command. If you use `ksh` and you want to use the `fc` command to invoke the CONVEX FORTRAN compiler instead of this editor, add the following aliases to your `.profile` file:

```
alias -x   fc='/usr/convex/fc'

alias -x   kfc='\fc'
```

The backslash preceding `fc` causes the shell to ignore the alias; `kfc` will thus invoke the `fc` editor. Note that you need to modify other aliases which reference the shell's `fc` command accordingly.

fcUnblock(1F)

Convert Cray format "blocked" unformatted sequential access file into a format the CONVEX FORTRAN compiler can read

Synopsis

`fcUnblock` <*input_file* >output_file

Description

The `fcUnblock` utility converts *input_file*, a Cray format "blocked" unformatted sequential access file, into *output_file*, a format that can be accessed by the CONVEX FORTRAN compiler through use of the `FORMAT=UNFORMATTED/CRAY` keyword definition in the `OPEN` statement. Use this utility only when the Cray file to be read is in the unformatted, "blocked" sequential access form. Pure data format files do not need to be converted. The `fcUnblock` utility reads from standard input and writes to standard output.

Notes

The CONVEX FORTRAN compiler accepts Cray unblocked sequential access format datafiles when the "CRAYUB" data format conversion is specified in an `OPEN` statement for a unit. For blocked sequential access files that have been processed by `fcUnblock`, and for unblocked direct access files, the "CRAY" data format conversion must be used. Refer to Chapter 7 in the *CONVEX FORTRAN Language Reference Manual*, "Binary Data File Format Conversions," for more information. A Program attempting to read a blocked binary data file will typically get an unexpected (premature) EOF.

Errors

Out of disk space. Because both the input and output files exist concurrently, substantial disk space may be required. Putting the input and output files on different file systems may be helpful.

fct(1F)

Convert a FORTRAN file for `fc`

Synopsis

`fct` [`-s`] [`-g`] [`-t`*n*] [*file*] ...

Description

`fct` converts FORTRAN source files to a form acceptable to `fc`. It replaces double quotes with single quotes, optionally expands tabs, and optionally removes sequence numbers from lines more than 72 characters long. When it removes a sequence number from a line, and the line does not appear to have any Hollerith fields, it also removes trailing blanks from the line.

`fct` accepts the following options:

Option	Meaning
`-s`	instructs `fct` not to strip sequence numbers.
`-g`	(guess) instructs `fct` not to strip sequence numbers from a source file if its first line is less than 72 characters long. This is useful for converting multiple source files, some of which may have sequence numbers.
`-t`*n*	instructs `fct` to expand tab characters to spaces, assuming tab stops every `n` spaces. `n` is optional; the default value is 8. Without this option, `fct` counts each tab as only one character when stripping sequence numbers.

If invoked with no files specified, `fct` accepts input from `stdin` and sends output to `stdout`. If invoked with a dash (-) as the second file, output is sent to `stdout`. Otherwise, `fct` overwrites each file.

Bugs

There are unusual situations in which `fct` fails to find Holleriths. It is useful, but simple.

See Also

`fc(1F)`

Notes

`fct` is an optional product; for more information, contact your CONVEX sales representative.

fcxref(1F)

FORTRAN cross-reference generator

Synopsis

`fc sourcefile ... -xra` [`-xrf` *datafile*]
`fcxref` [`-xrf` *datafile*] [*option* ...]

OR:

`fc` *sourcefile* ... `-xr` [*option* ...]

Description

`fcxref` generates a detailed report of references to each object in a FORTRAN source program. `fcxref` replaces the old cross-referencer `fxref`, which is no longer supported. Arguments whose names end with `.f` or `.FOR` are taken to be FORTRAN source programs. The following options are associated with `fcxref`:

Option	Meaning
`-xr`	Invokes cross-referencer and generates report at compile time. Automatically deletes cross-referencer data file after compilation. Use only on `fc` command line.
`-xra`	Invokes cross-referencer and generates data file but does not generate report. Used for globally cross-referencing several FORTRAN source files. Use only on `fc` command line.
`-xrd`	Used only when `fcxref` is run standalone. Deletes cross-referencer data file after report is generated. By default, `fcxref` saves the data file.
`-xrf` *datafile*	Use *datafile* as the cross-referencer data file instead of the default, `.fcxrefData`.

`-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 list, ordered by member offsets
  8: differing members list, ordered by member names

  If *n* = 0, no common block reports are generated. Default is *n* = 8.

`-xrr` *file*
: Sends the cross-reference report to *file* instead of the default, `stdout`.

`-iw` *n*
: Sets 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*.

`-pw` *n*
: Sets a desired page width of *n* columns per line. Data that exceeds this width is wrapped to the next line. Default is *n* = 80.

`-pl` *n*
: Sets a default maximum page length of *n* lines per page. When *n*-1 lines have been printed in the report, an ASCII form feed followed by a page header is automatically inserted. Default is *n* = 60. Note that `-sl` is not supported by `fcxref`; similar functionality is provided by `-LST`. Refer to the `fc(1F)` man page for more information.

Format

Cover Page:	lists date, time and parameter values.
Routine Reports:	one for each routine; ordered alphabetically by routine name. Lists routine name, kind (subroutine, function, main) and source file it resides in, followed by symbol, structure (if VMS structures used) and common block summary reports. Objects listed in the left column of the summary reports can be labels, variables, parameters, block data names, routine names, VMS structures (prefixed with `$`), or common blocks (prefixed with _). The right column specifies object type, offsetting information for array or record members, common block name if the object resides in a common block, and line number references in the form `<line number><include table index><context operator>`. The include table index refers to the Include File Reference Table (see below) and is present only if the object is contained in an include file. The context operator is defined as follows: <blank>: object referenced #: routine name appears in header line =: object assigned or defined *: object described
Caller/Callee Cross Reference:	lists routine's subroutine, function or intrinsic callers and callees.
Composite Common Block Reports:	for each common block, lists block name and maximum size followed by each member's name, offset and declaring routine. Use `-xrg` to specify the report ordering and format; see above.
Include File Reference Table:	provides a numbered list of include files referenced in the report. The `include table index` mentioned under Routine Reports refers to this list.

fcxref(1F)

Table of Contents:	lists subrouines, functions and common blocks and the page numbers on which they appear, ordered by object name.

Files

file.[f,FOR]	input file for `fc`
`.fcxrefData`	cross-referencer data file; rename with `-xrf`

See Also

`fc(1F), fxref(1F)`
CONVEX FORTRAN Language Reference Manual
CONVEX FORTRAN User's Guide

fpr(1F)

Print FORTRAN file

Synopsis

```
fpr
```

Description

`fpr` is a filter that transforms files formatted according to FORTRAN's carriage control conventions into files formatted according to UNIX line printer conventions.

`fpr` copies its input onto its output, replacing the carriage control characters with characters that produce the intended effects when printed using `lpr (1)`. The first character of each line determines the vertical spacing:

Character	Vertical spacing before printing
<blank>	One line
0	Two lines
1	To first line of next page
+	No advance
$	No carriage return

A blank line is treated as if its first character is a blank. A blank that appears as a carriage control character is deleted. A zero is changed to a newline. A one is changed to a form feed. The effects of a + are simulated using backspaces. A `$` starts output at the beginning of the next line and suppresses carriage return at the end of the line.

Examples

```
% a.out | fpr | lpr

% fpr < fc.output | lpr
```

Bugs

Results are undefined for input lines longer than 170 characters.

Notes

UNIX is a registered trademark of UNIX System Laboratories, Inc.

`fpr` is an optional product; for more information, contact your CONVEX sales representative.

fsplit(1F)

Split a multi-routine FORTRAN file into individual files

Synopsis

`fsplit` [`-72`] [`-e` *routine*] ... [*file*]

Description

`fsplit` reads FORTRAN source code from either a file or the standard input stream. It splits the input into a separate file for each program unit (i.e. function, subroutine, block data, or program). Each new file is named *name*`.f`, where *name* is the name of the program unit. Files containing unnamed block common or main programs are named `blkdata`*NNN*`.f` or `main`*NNN*`.f`, where *NNN* is a number chosen to make the filename unique. If there is an error in classifying a program unit, or if *name*`.f` already exists, the program unit is put in a file named zzz*NNN*`.f`. The command `-line` options are:

Option	Meaning
`-72`	process only the first 72 columns of each line. `fsplit` ignores characters beyond column 72 when this flag is specified, but it does not remove them from the file.
`-e` *routine*	extract only the named routine. Multiple `-e` options may be given. If no `-e` options are specified, all routines are extracted.

Diagnostics

If names specified via the `-e` option are not found, a diagnostic is written to `standard error`.

Bugs

`fsplit` assumes the subprogram name is on the first noncomment line of the subprogram unit. Nonstandard source formats may cause confusion. It is hard to use `-e` for unnamed main programs and block data subprograms because you must predict the created file name.

Notes

`fsplit` is an optional product; for more information, contact your CONVEX sales representative.

fxref(1F)

FORTRAN cross-reference generator

Synopsis

`fxref` [*option* ...] *file* ...

Description

`fxref` generates identifier cross-reference tables for FORTRAN source programs. `fxref` has been replaced by `fcxref` in CONVEX FORTRAN Version 7.0. `fxref` is no longer supported and will eventually be removed from the compiler. See the `fcxref`(1F) man page for more information on the new cross-referencer.

Arguments whose names end with `.f` or `.FOR` are taken to be FORTRAN source programs.

To use `fxref`, you must compile with the `-xro` or `-xr1` compiler options. `fxref` allows the following options:

Option	Meaning
`-dc`	Specifies that a line with a `D` in column 1 is to be processed as code rather than treated as a comment line. This provides a convenient way to keep debugging code in a source file.
`-fpp`	Causes source files to be processed by the FORTRAN preprocessor before being analyzed.
`-i` *n*	Sets the default size of integer and logical variables, in bytes. *n* may be 2, 4, or 8. If the `-i` flag is not specified, the default size will be 4.
`-iw` *n*	Controls the column width for identifiers. Can take values ranging from 8 to 32. Default value is 16. If this value is changed, the column numbers below are adjusted accordingly.
`-r` *n*	Sets the default size of real variables, in bytes. *n* may be 4 or 8. If the `-r` flag is not specified, the default size will be 4.
`-sl`	Produces a source listing with line numbers, preceding the cross-reference table.
`-pw` *n*	Controls the logical page width used by the output formatter. Default value is 132.

`-vn`	Identifies fxref version. Output goes to `stderr`.
`-xr1`	Put all objects (variables, arrays, etc.) into 1 big table, rather than print a separate table for each class of objects.
`-72`	Truncate source at column 72.

Format

Columns 1-16: identifier name

Columns 18-33: name of `PROGRAM`, `FUNCTION`, `SUBROUTINE`, or `BLOCK DATA` program that the identifier was found in.

Colummns 35-42: data type of identifier. The first character specifies the type:

R - `REAL`
I - `INTEGER`
L - `LOGICAL`
C - `CHARACTER`
Z - `COMPLEX`
<blank> - none.

The second character is a `*`. The remaining characters specify the width in bytes.

Column 44-46: object class:

ARY - array
BLK - block data
COM - common block
DO - do loop head
ENT - entry point
EXT - external
FUN - function
ITR - intrinsic
LAB - statement label
NML - namelist
PAR - parameter
PRG - program
STF - statement function
SUB - subroutine
VAR - variable

Remaining columns: The rest of the line gives the line number and usage class of each reference. Usage classes are denoted by the following characters:

- d - declared (`DIMENSION`, `EQUIVALENCE`, `COMMON`, <type> statement)
- i - initialized (`DATA`, `PARAMETER`)
- a - assigned
- p - passed as an argument to a function or subroutine
- blank - referenced

Files

`/usr/lib/fxnames`	generates file of name references
`/usr/lib/fxprint`	generates final output file

See Also

CONVEX FORTRAN Language Reference Manual
CONVEX FORTRAN User's Guide

Notes

References that appear in an included file appear on separate lines with the include file name right justified to column 46.

Names longer than 32 characters are truncated. If two names differ only after the first 32 characters, they are treated as the same identifier.

Processing syntactically invalid programs gives unpredictable results.

intro(3F)

Introduction to FORTRAN library functions

Description

This section describes those functions that are in the FORTRAN utility library. The functions listed here provide an interface from `fc` programs to the system in the same manner as the C library does for C programs. They are automatically loaded as needed by the FORTRAN compiler `fc`.

Functions

Name	Description
abort	terminate abruptly with memory image
access	determine accessability of a file
alarm	execute a subroutine after a specified time
aset	allow FORTRAN asynchronous I/O
bessel	of two kinds for integer orders
chdir	change default directory
chkpnt	checkpoint a process or process family
chmod	change mode of a file
ctime	return system time
date	return current date
dffrac	return fractional accuracy of double float
dflmax	return maximum positive double float
dflmin	return minimum positive double float
drand	return random values
dtime	return elapsed execution time
erf	returns error function
erfc	returns 1-error function

errsns	get system error messages
errtrap	trap arithmetic errors (newer)
etime	return elapsed execution time
exit	terminate process with status
fdate	return date and time in an ASCII string
ffrac	return fractional accuracy of single float
fgetc	get a character from a logical unit
flmax	return maximum positive single float
flmin	return minimum positive single float
flush	flush output to a logical unit
fork	create a copy of this process
fputc	write a character to a fortran logical unit
fseek	reposition a file on a logical unit
fstat	get file status
ftell	reposition a file on a logical unit
gerror	get system error messages
getarg	return command line arguments
getc	get a character from a logical unit
getcwd	get pathname of current working directory
getenv	get value of environment variables
getgid	get user or group ID of the caller
getlog	get user's login name
getpid	get process ID

getuid	get user or group ID of the caller
gmtime	return system time
hostnm	get name of current host
iargc	return command line arguments
idate	return date or time in numerical form
ierrno	get system error messages
inmax	return maximum positive integer value
intro	introduction to FORTRAN library functions
ioinit	change FORTRAN I/O initialization
irand	return random values
isatty	find name of a terminal port
itime	return date or time in numerical form
kill	send a signal to a process
libcfc	lists all supported Cray routines
libF90	lists all supported Fortran 90 routines
link	make a link to an existing file
lnblnk	tell about character objects
loc	return the address of an object
longjmp	jump to previously saved environment.
lstat	get file status
ltime	return system time
mvbits	move bits
perror	get system error messages

putc	write a character to a fortran logical unit
qsort	quick sort
rand	return random values
ran	return random values
rename	rename a file
restart	restart execution of a process or a process family
rindex	tell about character objects
secnds	return system time in secnds
setjmp	save environment for later longjmp.
signal	change the action for a signal
sleep	suspend execution for an interval
stat	get file status
stime	return system time
system	execute a command
taset	set/reset asynchronous status of the tape unit
tawait	wait for completion of pending I/O on the tape unit
tclose	close "raw" mode tape device
terase	write an interrecord gap to tape
time	return time in an ASCII string
toffl	rewind tape and put tape unit offline
topen	open "raw" mode tape device
topenreadonly	open "raw" mode tape device in readonly mode
traper	trap arithmetic errors

tread	read a physical record from tape
tretry	enable/disable retry on tape errors
trewind	rewind tape to the beginning
tskipf	reposition tape by files
tskipr	reposition tape by physical records
tstate	return the status of the tape unit
ttynam	find name of a terminal port
tweof	write an end-of-file record to tape
twrite	write a physical record to tape
unlink	remove a directory entry
wait	wait for a process to terminate

abort(3F)

Terminate abruptly with memory image

Synopsis

```
subroutine abort (string)
character*(*) string
```

Description

The `abort` subroutine cleans up the I/O buffers and then aborts producing a `core` file in the current directory. If `string` is given, it is written to logical unit 0 preceeded by `abort`.

Files

`/usr/lib/libU77.a`

access(3F)

Determine accessibility of a file

Synopsis

```
integer function access (name, mode)
character*(*) name, mode
```

Description

The access subroutine checks the given file, name, for accessibility with respect to the caller according to mode. The mode argument may include, in any order and in any combination, one or more of:

r	test for read permission
w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either argument is illegal, or if the file can not be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

Files

/usr/lib/libU77.a

See Also

access(2), perror(3F)

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Notes

The access subroutine is an optional product; for more information, contact your CONVEX sales representative.

alarm(3F)

Execute a subroutine after a specified time

Synopsis

```
integer function alarm (time, proc)
integer time
external proc
```

Description

The `alarm` subroutine arranges for subroutine `proc` to be called after `time` seconds. If `time` is 0, the alarm is turned off and no routine is called. The returned value is the time remaining on the last alarm.

Files

`/usr/lib/libU77.a`

See Also

`alarm(3C), sleep(3F), signal(3F)`

Bugs

The `alarm` and `sleep` routines interact. If `sleep` is called after `alarm`, the alarm process is never called. `SIGALRM` occurs at the lesser of the remaining alarm time or the sleep time.

Notes

The `alarm` subroutine is an optional product; for more information, contact your CONVEX sales representative.

aset(3F), await, aread, awrite

FORTRAN asynchronous I/O

Synopsis

```
integer function aset (lu, flags)
integer lu, flags

integer function await (lu, count)
integer lu, count

integer function aread (lu, pos, addr, count)
integer lu, pos, addr, count

integer function awrite (lu, pos, addr, count)
integer lu, pos, addr, count
```

Description

`aset`, `await`, `aread`, and `awrite` provide FORTRAN access to asynchronous I/O. You can access a file either synchronously via READ/WRITE statements or asynchronously via AREAD/AWRITE function calls, but you cannot access a file both ways without closing and reopening the file.

Asynchronous I/O allows the FORTRAN user to: 1) request I/O transfers, which are performed in parallel with user processing; 2) perform other operations not dependent on the data to be transferred; 3) synchronize with the data transfer; and 4) process the data transferred.

`aset` provides control over the asynchronous status of the file defined by logical unit `lu`. The flag's variable controls the asynchronous attributes of the file. The flags provided are added together and are:

`A_ASYNC '1'X` enables asynchronous I/O.

`A_FNCACHE '2'X` bypasses the system buffer cache when possible.

`A_SINGLE '4'X` allows only a single asynchronous operation per unit to be active at a time. An implied `await()` is performed before each `aread` and `awrite` request. The results of the `await()` call are discarded, except when an error occurs; when an error occurs, the I/O request aborts, and the error code is returned immediately.

`A_SIGNAL '8'X` generates a SIGIO signal on completion of all pending asynchronous operations for the specified unit. The signal can be caught by a user-specified FORTRAN subroutine; see `signal`(3F) for details.

`await` waits for completion of all pending asynchronous transfers for unit `lu` and returns the status of the transfers. `count` is set to the actual number of bytes transferred. If `count` is zero and the number of bytes requested for transfer is nonzero, then end-of-file has been reached. No attempt to detect end-of-file is made by `await;` this is left to the user. If an error occurs, a nonzero error code is returned. In the event of multiple errors, the first error detected is returned.

`aread` requests that a block of data of length `count` beginning at `addr` be read from logical unit `lu`. `pos` is the file positioning variable. If `pos` has a nonnegative value, then the file is positioned to begin reading at byte `pos`, where the first byte of the file is byte zero. If `pos` has a negative value, no file positioning is done. Control immediately returns to the user's program, which can continue to perform other processing while the actual data I/O takes place.

`awrite` requests that a block of data of length `count` beginning at `addr` be written to logical unit `lu`. `pos` is the file positioning variable. If `pos` has a nonnegative value, then the file is positioned to begin writing at byte `pos`, where the first byte of the file is byte zero. If `pos` has a negative value, no file positioning is done. Control immediately returns to the user's program, which can continue to perform other processing while the actual data I/O takes place.

Return Codes

Each routine returns zero for successful completion and a nonzero error code if an error occurs. Some of the possible error codes are:

`F_ERNOASYNC` asynchronous I/O cannot occur on this unit possibly because a synchronous event has already occurred on this unit

`F_ERNOSYNC` synchronous I/O cannot occur on this unit possibly because an asynchronous event has already occurred on this unit

See Also

`asiostat(2), fcntl(2), read(2), write(2), signal(3F)`

Notes

The routines `aset`, `await`, `aread`, and `awrite` should be explicitly declared to be type integer.

The following example illustrates an asynchronous routine:

```
      program main
      integer await, aset, aread, awrite
      external await, aset, aread, awrite
      real*8 amat(10240), bmat(10240)
      integer A_ASYNC
      parameter (A_ASYNC = '1'X)

      open(1, file='foo.dat', err=999, iostat=iret)
      iflag = A_ASYNC
      iret = aset(1, iflag)
      if (iret .NE. 0) goto 999
      do i = 1, 10240
          amat(i) = i
      enddo

c Write amat to file
      iret = awrite(1, 0, amat, 10240*8)
      if (iret .NE. 0) goto 999

c do other processing NOT USING amat
      do i = 1, 10240
          bmat(i) = -i
      enddo

c ready for another output request
c  NOTE: positioning of -1 means start writing where
c  we left off
      iret = awrite(1, -1, bmat, 10240*8)

c You can do more stuff here but don't modify amat or
c bmat

c sync with I/O request (we can't do anything until
c    we can modify amat and bmat)
c we will wait for BOTH I/O requests to complete
      iret = await(1, icnt)
      if (iret .NE. 0) goto 999

c If we don't write both arrays, then we have problems
      if (icnt .NE. 10240*8*2) goto 998
```

```
c done...
      close (1)
      stop

c error
998 print *, 'unexpected error: only wrote ', icnt, '
bytes.'
      close (1)
      stop

999 print *, 'error ', iret, ' occurred.'
      close (1)
      stop
      end
```

It should be noted that asynchronous I/O requires substantially more compute time for disk operations than for tape operations. The reason for this is that the current implementation of asynchronous I/O requires a minimum of four context switches per I/O operation. In the context of slow I/O devices such as tape, these switches form a minimal overhead. However, for disk I/O with large block transfers and multi-way striping, this overhead becomes significant because CONVEX disk I/O already uses buffered writes and read-ahead for disk operations. Assuming an adequate disk buffer cache (default is 10% of physical memory), normal FORTRAN I/O to disk gains most of the benefits of async I/O, without incurring the extra overhead.

`aset` is an optional product; for more information, contact your CONVEX sales representative.

bessel functions(3F)

Bessel functions of two kinds for integer orders

Synopsis

```
function besj0 (x)

function besj1 (x)

function besjn (n, x)

function besy0 (x)

function besy1 (x)

function besyn (n, x)

double precision function dbesj0 (x)
double precision x

double precision function dbesj1 (x)
double precision x

double precision function dbesjn (n, x)
double precision x

double precision function dbesy0 (x)
double precision x

double precision function dbesy1 (x)
double precision x

double precision function dbesyn (n, x)
double precision x
```

Description

These functions calculate `bessel` functions of the first and second kinds for real arguments and integer orders.

Diagnostics

Negative arguments cause `besy0, besy1,` and `besyn` to return a huge negative value. The system error code is set to EDOM (33).

Files

`/usr/lib/libU77.a`

See Also

`j0(3M), perror(3F)`

Notes

In order to link a program which calls any of these functions, the `-m` option, and either the `-lC1_old` or the `-lC2_old` option must be used when linking.

`bessel` is an optional product; for more information, contact your CONVEX sales representative.

chdir(3F)

Change default directory

Synopsis

```
integer function chdir (dirname)
character*(*) dirname
```

Description

The default directory for creating and locating files is changed to `dirname`. Zero is returned if successful; an error code is returned otherwise.

Files

`/usr/lib/libU77.a`

See Also

`chdir(2), cd(1), perror(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>

Use of this function may cause `inquire` by unit to fail.

Notes

`chdir` is an optional product; for more information, contact your CONVEX sales representative.

chkpnt(3F)

Checkpoint a process or process family

Synopsis

```
integer function chkpnt (class, pid, name, options,
                         signo)
integer class
integer pid
character*(*) name
integer options
integer signo
```

Description

The `chkpnt` function creates one or more files that contain all the process state information necessary to restart a process or process hierarchy. One checkpoint file is created for each successfully checkpointed process. The arguments have the following meanings:

`class`	The type of checkpoint to perform.
`pid`	The identifier of the target process to checkpoint. A `pid` of 0 indicates the calling process.
`name`	The name of the checkpoint files. By default, checkpoint files are created with the name `comm.pid`, where `comm` is the trailing component of the command name of the process and `pid` is the process id. If the argument `name` is non-null, the checkpoint files will be named `name.pid`.
`options`	Optional actions to be performed by the `chkpnt` routine.
`signo`	Depending on the value of the `flags` argument, the `signo` parameter may be interpreted as a signal to send to the target process if the checkpoint is successful. See `chkpnt`(3) for a more complete description of the arguments to `chkpnt`.

Return Value

`chkpnt` returns 0 if successful; otherwise an error code is returned.

Files

`/usr/lib/libc_old.a`

See Also

`chkpnt(3), restart(3), restart(3f)`

chmod(3F)

Change mode of a file

Synopsis

```
integer function chmod (name, new_mode)
character*(*) name, new_mode
```

Description

This function changes the filesystem `mode` of file `name` . `new_mode` can be any specification recognized by `chmod`. `name` must be a single pathname.

The normal returned value is 0. Any other value is a system error number.

Files

`/usr/lib/libU77.a`
`/bin/chmod` exec'ed to change the mode.

See Also

`chmod(1)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>

Notes

`chmod` is an optional product; for more information, contact your CONVEX sales representative.

date(3F)

Return date in an ASCII string

Synopsis

```
subroutine date (buf)
character *(*) buf
```

Description

`date` returns the current date in a 9-character ASCII string. The string format is: `dd-mmm-yy`. This string gives the day of the month, dd (1-31), the month of the year, mmm (e.g., JAN), and the year-1900, yy. `buf` is any 9-byte variable or character string.

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2), ctime(3), etime(3F), fdate(3F), idate(3F), secnds(3F), stime (3F), time(3F)`

Notes

`date` is an optional product; for more information, contact your CONVEX sales representative.

erf(3F), erfc, derf, derfc

Error functions

Synopsis

```
real function erf(x)
real x

real function erfc(x)
real x

double precision function derf(x)
double precision x

double precision function derfc(x)
double precision x
```

Description

`Erf(x)` returns the error function of `x`; where

erf(x) = 2/sqrt(pi)*integral from 0 to x of exp(-t*t) dt.

erfc(x) returns 1.0-erf(x).

The entry for `erfc` is provided because of the extreme loss of relative accuracy if erf(x) is called for large x and the result subtracted from 1. (e.g. for x = 10, 12 places are lost).

`derf` and `derfc` are the double precision versions of `erf` and `derf` respectively.

Files

`/usr/lib/libU77.a`

Notes

`erf` is an optional product; for more information, contact your CONVEX sales representative.

errsns(3F)

Get error information

Synopsis

```
subroutine errsns (fnum,rmssts,rmsstv,iunit,condval)
integer fnum,rmssts,rmsstv,iunit,condval
```

Description

`errsns` returns information about the error that has most recently occurred during program execution. `fnum` is set to the current value of the system error register. The arguments `rmssts, rmsstv, iunit,` and `condval` are not used.

Files

`/usr/lib/libU77.a`

See Also

`perror(3f), intro(2), perror(3)`

Notes

`errsns` is an optional product; for more information, contact your CONVEX sales representative.

errtrap(3F)

Trap arithmetic errors

Synopsis

```
integer function errtrap (mask)
```

Description

By default, floating-point underflow, integer overflow, and intrinsic errors are not trapped during program execution. However, integer divide by zero, and the floating point trap (which includes floating point overflow, reserved operand fault, and floating point divide by zero) are enabled by default during program execution.

This routine changes the setting of the floating-point underflow, integer overflow, integer divide by zero, and floating point traps by setting or resetting the status bits in the process status word. The trap-enable bits are inherited from the calling subprogram, and the previous state is restored on return.

If any of the errors occurs and trapping is enabled, signal SIGFPE is sent to the process. (See `signal (3C).`) By default, the occurrence of an intrinsic error (e.g., square root of a negative real value) does not terminate execution of the program. Setting the intrinsic error trap causes the program to terminate on the first intrinsic error encountered. The intrinsic error trap value applies to the entire program.

The argument to `errtrap` is produced by summing the appropriate flags below:

Value	Meaning
'01'X	trap integer overflow
'02'X	trap floating underflow
'04'X	trap intrinsic errors
'08'X	trap floating point. (FDZ,OV,RO)
'10'X	trap integer divide by zero

As an example, `i=errtrap('3'X)` would enable integer overflow and floating underflow, and would disable intrinsic errors, integer divide by zero, and floating point trap.

The previous value of these flags is returned.

Files

`/usr/lib/libU77.a`
CONVEX FORTRAN User's Guide, Chapter 8

See Also

`signal(3C), signal(3F)`

Notes

`errtrap` is an optional product; for more information, contact your CONVEX sales representative.

etime(3F), dtime

Return elapsed execution time

Synopsis

```
function etime (tarray)
real tarray(2)

function dtime (tarray)
real tarray(2)
```

Description

These two routines return elapsed runtime in seconds for the calling process. `dtime` returns the elapsed time since the last call to `dtime`, or the start of execution on the first call.

The argument array returns user time in the first element and system time in the second element. The function value is the sum of user and system time.

The resolution of all timing is one microsecond.

Files

`/usr/lib/libU77.a`

See Also

`getrusage(2), date(3F), fdate(3F), idate(3F), secnds(3F), stime(3F), time (3F)`

Notes

`etime` is an optional product; for more information, contact your CONVEX sales representative.

exit(3F)

Terminate process with status

Synopsis

```
subroutine exit (status)
integer status
```

Description

`exit` flushes and closes all the process's files, and notifies the parent process if it is executing a `wait`. The low-order 8 bits of `status` are available to the parent process. (Therefore `status` should be in the range 0 to 255.) If `exit` is called without arguments, `status` is defaulted to zero.

This call never returns.

The C function `exit` may cause cleanup actions before the final `sys exit`.

Files

`/usr/lib/libU77.a`

See Also

`exit(2), fork(2), fork(3F), wait(2), wait(3F)`

Notes

`exit` is an optional product; for more information, contact your CONVEX sales representative.

fdate(3F)

Return date and time in an ASCII string

Synopsis

```
subroutine fdate (string)
character*(*) string

character*(*) function fdate()
```

Description

`fdate` returns the current date and time as a 24-character string in the format described under `ctime(3)`. Neither newline nor NULL is included.

`fdate` can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24    fdate
external        fdate

write(*,*) fdate()
```

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2), ctime(3), time(3F), itime(3F), idate(3F), secnds(3F), stime(3F), date(3F)`

Notes

`fdate` is an optional product; for more information, contact your CONVEX sales representative.

flmin(3F), flmax, ffrac, dflmin, dflmax, dffrac, qflmin, qflmax, qffrac, inmax

Return extreme values

Synopsis

```
function flmin()

function flmax()

function ffrac()

double precision function dflmin()

double precision function dflmax()

double precision function dffrac()

real*16 function qflmin()

real*16 function qflmax()

real*16 function qffrac()

function inmax()
```

Description

Functions `flmin` and `flmax` return the minimum and maximum positive floating-point values respectively. Functions `dflmin` and `dflmax` return the minimum and maximum positive double-precision floating-point values. Functions `qflmin` and `qflmax` return the minimum and maximum positive real*16 precision floating-point values. Function `inmax` returns the maximum positive integer value.

The functions `ffrac` `dffrac` and `qffrac` return the fractional accuracy of single, double and real*16 precision floating point numbers respectively. These are the smallest numbers that can be added to 1.0 without being lost.

These functions can be used by programs that must scale algorithms to the numerical range of the processor.

Files

`/usr/lib/libU77.a`

Notes

`flmin` is an optional product; for more information, contact your CONVEX sales representative.

flush(3F)

Flush output to a logical unit

Synopsis

```
subroutine flush (lunit)
```

Description

`flush` causes the contents of the buffer for logical unit `lunit` to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

Files

`/usr/lib/libU77.a`

See Also

`fclose(3S)`

Notes

`flush` is an optional product; for more information, contact your CONVEX sales representative.

fork(3F)

Create a copy of this process

Synopsis

```
integer function fork()
```

Description

`fork` creates a copy of the calling process. The original process is known as the parent process and the new one is known as the child process. The only distinction between the two is that the value returned to the parent is the process id of the child while the value returned to the child process is zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and is the negation of the system error code. (See `perror`(3F).)

A corresponding `exec` routine has not been provided because there is no satisfactory way to retain open logical units across the exec. The usual function of `fork/exec`, however, can be performed using `system(3F)`.

Files

`/usr/lib/libU77.a`

See Also

`fork(2), wait(3F), kill(3F), system(3F), perror(3F)`

Notes

`fork` is an optional product; for more information, contact your CONVEX sales representative.

fseek(3F), ftell

Reposition a file on a logical unit

Synopsis

```
integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)
```

Description

`lunit` must refer to an open logical unit. `offset` is an offset in bytes relative to the position specified by `from`. Valid values for `from` are:

Value	Meaning
0	beginning of the file
1	the current position
2	the end of the file

The value returned by `fseek` is 0 if successful; a system error code otherwise. (See `perror`(3F).)

`ftell` returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and is the negation of the system error code. (See `perror`(3F).)

Files

`/usr/lib/libU77.a`

See Also

`fseek(3S), perror(3F)`

Notes

`fseek` is an optional product; for more information, contact your CONVEX sales representative.

getarg(3F), iargc

Return command line arguments

Synopsis

```
subroutine getarg (k, arg)
character*(*) arg

function iargc ()
```

Description

A call to `getarg` returns the k*th* command line argument in character string `arg`. The 0*th* argument is the command name.

`iargc` returns the index of the last command line argument.

Files

`/usr/lib/libU77.a`

See Also

`getenv(3F), execve(2)`

Notes

`getarg` is an optional product; for more information, contact your CONVEX sales representative.

getc(3F), fgetc

Get a character from a logical unit

Synopsis

```
integer function getc (char)
character char

integer function fgetc (lunit, char)
character char
```

Description

These routines return the next character from a file associated with a FORTRAN logical unit, bypassing normal FORTRAN I/O. `getc` reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occurred on the read; -1 indicates end-of-file was detected. A positive value is either a ConvexOS system error code or a FORTRAN I/O error code. (See `perror`(3F).)

Files

`/usr/lib/libU77.a`

See Also

`getc(3S), intro(2), perror(3F)`

Notes

`getc` is an optional product; for more information, contact your CONVEX sales representative.

getcwd(3F)

Get pathname of current working directory

Synopsis

```
integer function getcwd (dirname)
character*(*) dirname
```

Description

The pathname of the default directory for creating and locating files is returned in `dirname`. The value of the function is zero if successful; an error code otherwise.

Files

`/usr/lib/libU77.a`

See Also

`chdir(3F), perror(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>

Notes

`getcwd` is an optional product; for more information, contact your CONVEX sales representative.

getenv(3F)

Get value of environment variables

Synopsis

```
subroutine getenv (ename, evalue)
character*(*) ename, evalue
```

Description

`getenv` searches the environment list (see `environ(7)`) for a string of the form `ename = value` and returns value in evalue if such a string is present; otherwise getenv fills evalue with blanks.

Files

`/usr/lib/libU77.a`

See Also

`environ(7), execve(2)`

Notes

`getenv` is an optional product; for more information, contact your CONVEX sales representative.

getlog(3F)

Get user's login name

Synopsis

```
subroutine getlog (name)
character*(*) name

character*(*) function getlog()
```

Description

`getlog` returns the user's login name. It returns blanks if the process is running detached from a terminal.

Files

`/usr/lib/libu77.a`

See Also

`getlogin(3)`

Notes

`getlog` is an optional product; for more information, contact your CONVEX sales representative.

getpid(3F)

Get process ID

Synopsis

```
integer function getpid()
```

Description

`getpid` returns the process ID number of the current process.

Files

`/usr/lib/libU77.a`

See Also

`getpid(2)`

Notes

`getpid` is an optional product; for more information, contact your CONVEX sales representative.

getuid(3F), getgid

Get user or group ID of the caller

Synopsis

```
integer function getuid()

integer function getgid()
```

Description

These functions return the real user or group ID of the user of the process.

Files

`/usr/lib/libU77.a`

See Also

`getuid(2)`

Notes

`getuid` is an optional product; for more information, contact your CONVEX sales representative.

hostnm(3F)

Get name of current host

Synopsis

```
integer function hostnm (name)
character*(*) name
```

Description

This function puts the name of the current host into character string `name`. The return value should be 0; any other value indicates an error.

Files

`/usr/lib/libU77.a`

See Also

`gethostname(2)`

Notes

`hostnm` is an optional product; for more information, contact your CONVEX sales representative.

idate(3F), itime

Return date or time in numerical form

Synopsis

```
subroutine idate (iarray)
integer iarray(3)

subroutine itime (iarray)
integer iarray(3)
```

Description

`idate` returns the current date in `iarray`. The order is: day, month, year. Month is in the range 1-12. Year is >= 1969.

`itime` returns the current time in `iarray`. The order is: hour, minute, second.

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2)`, `ctime(3)`, `date(3F)`, `etime(3F)`, `fdate(3F)`, `secnds(3F)`, `stime(3F)`, `time(3F)`

Notes

`idate` is an optional product; for more information, contact your CONVEX sales representative.

ioinit(3F)

Change FORTRAN I/O initialization

Synopsis

```
subroutine ioinit (cctl, bzro, apnd, opst, prefix,
                   vrbose)
logical cctl, bzro, apnd, opst, vrbose
character*(*) prefix
```

Description

This routine initializes several global parameters in the FORTRAN I/O system, and attaches externally defined files to logical units at run time. The effect of the flag arguments applies to logical units opened after `ioinit` is called. The exception is the preassigned units, 5 and 6, to which `cctl` and `bzro` apply without an OPEN statement after the call to `ioinit` is executed.

By default, carriage control is not recognized on any logical unit. If `cctl` is .TRUE., then carriage control is recognized on formatted output to all logical units except unit 0, the diagnostic channel. Otherwise, the default is restored.

The interpretation of carriage control via `fpr`(1), rather than the `cctl` parameter, is recommended.

By default, trailing and embedded blanks in input data fields are ignored. If `bzro` is .TRUE., then such blanks are treated as zeros. Otherwise, the default is restored.

By default, a file that is opened has an lqunknownrq status; that is, either an existing file is opened or a new file is created. If `opst` is .TRUE., the default file status is lqnewrq (and the file must not exist).

By default, the marker for each file opened for sequential access is positioned at the beginning of the file.

If `apnd` is .TRUE., the marker for files opened subsequently on any logical unit is positioned at the end of file upon opening, so that a write will append to the existing data. A value of `.false.` restores the default behavior. If the argument `prefix` is a nonblank string, then names of the form `prefix`NNN are sought in the program environment. The value associated with each such name found is used to open logical unit NNN for formatted sequential access.

For example, if the FORTRAN program `myprogram` includes the statement:

```
     call ioinit (.true., .false., .false.,.false.,
    1 'FORT', .false.)
```

then the following sequence:

```
% setenv FORT001 mydata
% setenv FORT112 myresults
% myprogram
```

results in logical unit 1 opened to file `mydata` and logical unit 112 opened to file `myresults`. Both files are positioned at their beginning. Any formatted output has column 1 removed and interpreted as carriage control. Embedded and trailing blanks are ignored on input.

If the argument `vrbose` is .TRUE., then `ioinit` reports on its activity. The effect of:

```
     call ioinit (.true., .true., .false.,.true., '',
    1 .false.)
```

can be achieved without the actual call by including `-F66` on the `fc` command line. This gives carriage control on all logical units except 0, causes files to be opened at their beginning, and causes blanks to be interpreted as zeros.

The internal flags are stored in a labeled common block with the following definition:

integer*2 ieof, ictl, izro, ist
common /ioiflg/ ieof, ictl, izro, ist

Files

`/usr/lib/libI77.a`	fc I/O library
`/usr/lib/libI66.a`	sets older FORTRAN I/O modes
`/usr/lib/libU77.a`	FORTRAN utility library

See Also

`getarg(3F), getenv(3F)`
CONVEX FORTRAN User's Guide

Bugs

`prefix` can be no longer than 30 characters. A pathname associated with an environment name can be no longer than 255 characters. The "+" carriage control does not work.

Notes

Convex FORTRAN now automatically associates environment variables of the form FORnnn (where the n's represent decimal digits of the unit number) with FORTRAN logical units. Files explicitly opened (by an OPEN statement) may be associated with any environment variable: the NAME= value from the OPEN statement is interpreted as an environment variable if possible. These features supercede most uses of `ioinit`.

`ioinit` is an optional product; for more information, contact your CONVEX sales representative.

kill(3F)

Send a signal to a process

Synopsis

```
function kill (pid, signum)
integer pid, signum
```

Description

`pid` must be the process id of one of the user's processes. `signum` must be a valid signal number (see sigvec(2)). The returned value is 0 if successful; an error code otherwise.

Files

`/usr/lib/libU77.a`

See Also

`kill(2), sigvec(2), signal(3F), fork(3F), perror(3F)`

Notes

`kill` is an optional product; for more information, contact your CONVEX sales representative.

libcfc.a(3F)

Cray FORTRAN library

Description

This section describes routines appearing in the Cray FORTRAN library. It also describes Cray intrinsics supported by CONVEX in Cray mode.

Functions

The routines listed below reside in `libcfc.a`.

Name	Description
cft$bool	Return 1.0e0 is logical argument is true; 0.0e0 otherwise.
cft$dprod	Return real product of two real args.
clock	Return current time.
date	Return current date.
dump(F,L,T)	Dump memory contents from F to L using T (T=1 for real, 2 for decimal, 3 for octal), then exit program.
etime	Return elapsed execution time.
findch(PAT,LEN,STR, ST,N,IFND)	Search STR of length N for string PAT starting at ST in STR and return in IFND (else IFND=0).
gather(N,A,B,IND)	Gathers N elements from B (specified by vector IND) into A.
hpalloc(IADDR,LEN, ERR,AB)	Allocate LEN words, return address in IADDR, set ERR<>0 if error, exit on error if AB is TRUE or nonzero.
hpcheck	Sets ERRCODE to zero and returns.
hpclmove(ADDR, LEN,STAT,AB)	Change size of previously allocated heap object to LEN; STAT<0 if error, 1 if object relocated, 0 otherwise; exit on error if abort is TRUE.
hpdeallc	Deallocates a previously allocated heap object.
hpdump	For compatability only; returns.

hpnewlen(ADDR, LEN,STAT,AB)	Returns 0 if new size<old size, 1 otherwise; STAT=negative on errors, 0 if LEN<old length, 1 otherwise; exit on error if AB=TRUE.
hpshrink	For compatability only; returns.
iceil(I,J)	Returns integer >= I/J.
igtbyt(STR,I)	Returns Ith byte of STR (bytes numbered from 0).
ihplen(ADDR,ERR, AB)	Returns length of heap object starting at ADDR; ERR always set to 0.
ihpstat(I)	For compatability only; for I=1 returns current heap length in words; I=10 or 12 returns large positive number; I=11 returns 0.
ilsum(N,L,INC)	Examines N values spaced INC words apart in a LOGICAL, REAL or INTEGER array L and returns number of TRUE or negative values.
int24(I)	Returns I; for compatability only.
jdate	Returns current Julian date.
komstr(SRC,ISB, NUM,DST,IDB)	Compares the first NUM bytes in the strings SRC and DST, starting at byte offsets ISB and IDB respectively; returns 0 if equal, -1 if first non-matching char in SRC<respective char in DST, 1 otherwise.
lint(I)	For compatibility only; returns I.
mvc(SRC,ISB,DST, IDB,NUM)	Move NUM bytes from SRC (offset by ISB) to DST (offset by IDB).
pack(PK,N,UPK,NW)	Packs NW words from UPK, using N rightmost bits from each word, and stores result to PK.
putbyt(STR,I,VAL)	Sets the Ith byte of STR to VAL.
rbn(BLANKS)	Converts trailing blanks in BLANKS to nulls; BLANKS is assumed to be a word.
rnb(NULLS)	Like rbn, but converts trailing NULLS to BLANKS.

scopy	Copy a real vector.
sdot	Returns the dot product of two real vectors.
second(R)	Returns number of seconds elapsed since start of job.
ssum	Sums the elements of a real vector.
strmov(SRC,ISB, NUM,DST,IDB)	Copies NUM bytes from STR to DEST; STR and DEST are offset by ISB and IDB respectively.
timef(R)	Returns ellapsed time in milliseconds since the last call to timef.
tr(SRC,ISB,NUM, TBL)	Translates NUM bytes (in place) of SRC (offset by ISB) using translation table TBL.
unpack(PK,N,UPK, NW)	Unpacks PK into NW words of UPK placing contiguous NBITS bits of PK into NBITS rightmost bits of next unpacked word.

Cray Intrinsics

The routines listed below are available as intrinsics in Cray mode.

Name	Description
and	Logical product.
compl	Logical compliment.
cot	Cotangent.
csmg	Conditional scalar merge.
cvmgm	Conditional vector merge/minus.
cvmgn	Conditional vector merge/nonzero.
cvmgp	Conditional vector merge/positive or zero.
cvmgt	Conditional vector merge/true.
cvmgz	Conditional vector merge/zero.
dcot	Double precision cotangent.

eqv	Equivalence.
leadz	Tallies the number of leading zeros.
loc	Returns memory address of specified variable or array.
mask(N)	Returns a word with the leftmost N bits set if N<=64; returns a word with the (128-N) rightmost bits set if N>64.
movbit(S,I,N,D,J)	Moves N bits from I to J, offsetting I by S bits, and D by J bits; S and D may be arrays.
neqv	Logical difference.
numarg	Number of arguments.
or	Logical sum.
popcnt	Population count.
poppar	Population parity count.
ranf	Obtain random number.
ranget	Obtain random seed.
ranset	Set random seed.
shift	Circular shift.
shiftl	Logical shift left.
shiftr	Logical shift right.
xor	Logical difference.

libF90.a(3F)

Fortran 90 library

Description

This section describes intrinsics appearing in the Fortran 90 library. These routines represent a limited subset of Fortran 90 intrinsics. They can only be accessed when the `-f90` option is specified on the command line.

Functions

The routines listed below reside in `libF90.a`.

Name	Description
DOT_PRODUCT (VECTA,VECTB)	Dot product of two rank-one arrays.
MATMUL(MATA, MATB)	Matrix multiplication.
ALL(MASK,DIM)	True if all values are true; DIM is optional.
ANY(MASK, DIM)	True if any value is true; DIM opional.
COUNT(MASK,DIM)	Number of true elements in array; DIM is optional.
MAXVAL(ARRAY, DIM,MASK)	Maximum value in ARRAY; optional DIM, MASK.
MINVAL(ARRAY, DIM,MASK)	Minimum value in ARRAY; optional DIM, MASK.
PRODUCT(ARRAY, DIM,MASK)	Product of array elements; optional DIM, MASK.
SUM(ARRAY,DIM, MASK)	Sum or array elements; optional DIM, MASK.
MERGE(TSOURCE, FSOURCE, MASK)	Merge under mask.
PACK(ARRAY,MASK, VECTOR)	Pack ARRAY into an array of rank one under MASK.
SPREAD(SOURCE, DIM,NCOPIES)	Replicates array by adding a dimension.

UNPACK(VECTOR, MASK,FIELD)	Unpack ARRAY of rank one into an array under MASK.
TRANSPOSE(MATRIX)	Transpose of an array of rank two.
MAXLOC(ARRAY, MASK)	Location of a maximum value in an array; optional MASK.
MINLOC(ARRAY, MASK)	Location of a minimum value in an array; optional MASK.

Notes

CONVEX FORTRAN's implementation of these Fortran 90 intrinsics does not allow for `KEYWORD =` syntax in the above functions. If you want to omit an optional argument that is not the last argument in the argument list, you must supply a `0` in its place. For example, to omit the `DIM` argument from a call to `SUM` when summing elements of array `A` using mask `B`, you would use the following syntax:

```
C = SUM(A,0,B)
```

To omit optional aguments that are last in the argument list, no `0` is required; you can leave the argument out.

link(3F)

Make a link to an existing file

Synopsis

```
function link (name1, name2)
character*(*) name1, name2

integer function symlnk (name1, name2)
character*(*) name1, name2
```

Description

`name1` must be the pathname of an existing file. `name2` is a pathname to be linked to file `name1`. `name2` must not already exist. The returned value is 0 if successful; a system error code otherwise.

`symlnk` creates a symbolic link to `name1`.

Files

`/usr/lib/libU77.a`

See Also

`link(2)`, `symlink(2)`, `perror(3F)`, `unlink(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Notes

`link` is an optional product; for more information, contact your CONVEX sales representative.

loc(3F)

Return the address of an object

Synopsis

```
function loc (arg)
```

Description

The returned value is the address of `arg`.

Files

`/usr/lib/libU77.a`

Notes

`loc` is an optional product; for more information, contact your CONVEX sales representative.

mvbits(3F)

Transfer bit fields

Synopsis

```
subroutine mvbits(m,i,len,n,j)
integer m,i,len,n,j
```

Description

`mvbits` 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). Other bits of the destination location and all of the bits of the source location remain unchanged. The values of i+len and j+len must be less than or equal to 32.

Files

`/usr/lib/libU77.a`

Notes

`mvbits` is an optional product; for more information, contact your CONVEX sales representative.

perror(3F), gerror, ierrno

Get system error messages

Synopsis

```
subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()
```

Description

`perror` writes a message to FORTRAN logical unit 0 appropriate to the last detected system error. `string` is written preceding the standard error message.

`gerror` returns the system error message in character variable `string`. `gerror` may be called either as a subroutine or as a function.

`ierrno` returns the error number of the last detected system error. This number 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 value is a more reliable indicator of what caused the error condition.

Files

`/usr/lib/libU77.a`

See Also

`intro(2), perror(3)`
CONVEX FORTRAN User's Guide, Appendix B

Bugs

`string` in the call to `perror` can be no longer than 127 characters.

The length of the string returned by `gerror` is determined by the calling program.

perror(3F), gerror, ierrno

Notes

ConvexOS system error codes are described in `intro(2)`. The FORTRAN I/O error codes and their meanings are:

Error Code	Meaning
100	error in format
101	illegal unit number
102	formatted io not allowed
103	unformatted io not allowed
104	direct io not allowed
105	sequential io not allowed
106	can't backspace file
107	off beginning of record
108	can't stat file
109	no * after repeat count
110	off end of record
111	truncation failed
112	incomprehensible list input
113	out of free space
114	unit not connected
115	read unexpected character
116	blank logical input field
117	new file exists
118	can't find lqoldrq file
119	unknown system error
120	requires seek ability

121	illegal argument
122	negative repeat count
123	illegal operation for unit
124	new record not allowed
125	numeric keyword variable overflowed
126	record number exceeds maximum records
127	file is read-only
128	variable record format not allowed
129	invalid record length
130	exceeds maximum number of open files
131	data type size too small for real
132	infinite loop in format
133	illegal recordtype
134	attempt to read nonexistant record in direct access file
135	attempt to reopen file with different unit number
136	incompatible format code and i/o list item type
137	unknown record length
138	synchronous i/o not allowed
139	synchronous i/o not allowed
140	incompatible format structure
141	namelist error
142	apparent recursive logical name definition
143	recursive input/output operation

144	out of free space, possibly due to unformatted I/O on a formatted file
145	Error in conversion of string to numeric
146	binary I/O data format conversion routine returned error
147	partial record I/O not supported (BUFFERIN/BUFFEROUT)

Notes

`perror` is an optional product; for more information, contact your CONVEX sales representative.

putc(3F), fputc

Write a character to a FORTRAN logical unit

Synopsis

```
integer function putc (char)
character char

integer function fputc (lunit, char)
character char
```

Description

These funtions write a character to the file associated with a FORTRAN logical unit bypassing normal FORTRAN I/O. `putc` writes to logical unit 6, normally connected to the control terminal output. The value of each function is zero unless some error occurred; a system error code otherwise. (See `perror`(3F).)

Files

`/usr/lib/libU77.a`

See Also

`putc(3S), intro(2), perror(3F)`

Notes

`putc` is an optional product; for more information, contact your CONVEX sales representative.

qsort(3F)

Quick sort

Synopsis

```
subroutine qsort (array, len, isize, compar)
external compar
integer*4 compar
```

Description

One-dimensional `array` contains the elements to be sorted. `Len` is the number of elements in the array. `isize` is the size of an element, typically:

Value	Data type
4	`integer` and `real`
8	`double precision` or `complex`
16	`double complex`
(length of character object)	`character` arrays

`compar` is the name of a user-supplied integer*4 function that determines the sorting order. This function is called with two arguments that are elements of `array`. The function must return:

Return value	Condition
negative	if arg 1 is considered to precede arg 2
zero	if arg 1 is equivalent to arg 2
positive	if arg 1 is considered to follow arg 2

On return, the elements of `array` sorted.

Files

`/usr/lib/libU77.a`

See Also

`qsort(3)`

Notes

`qsort` is an optional product; for more information, contact your CONVEX sales representative.

rand(3F), drand, qrand, irand, ran

Return random values

Synopsis

```
function rand (iflag)

double precision function drand (iflag)

real*16 function qrand (iflag)

function irand (iflag)

function ran (iseed)
```

Description

These functions use a linear congruential random-number generator to generate sequences of random numbers. If `iflag` is nonzero, it is used as a new seed for the random-number generator, and the random-number generator is initialized. If `iflag` is zero, the next random number is returned. `ran` modifies its argument, `iseed`, which should be used as the argument of subsequent calls to `ran`.

`irand` returns positive integers in the range 0 through 2147483647. `rand`, `drand`, `qrand`, and `ran` return values in the range 0.0 through 1.0.

Files

`/usr/lib/libU77.a`

Notes

`rand`, `irand`, `drand`, `qrand`, and `ran` use a linear congruential random-number generator of the form:

```
x(i+1) = x(i) * 69069 + 1.
```

This can cause an integer overflow.

`rand` is an optional product; for more information, contact your CONVEX sales representative.

rcvtir(3F), dcvtid, ircvtr, idcvtd

IEEE/native floating point mode conversion routines

Synopsis

```
real*4 function rcvtir(x)
real*4 x

real*8 function dcvtid(x)
real*8 x

real*4 function ircvtr(x)
real*4 x

real*8 function idcvtd(x)
real*8 x
```

Description

`rcvtir` returns the value of its single precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

`dcvtid` returns the value of its double precision native mode input in IEEE mode. A dirty zero (i.e. non-zero fractional part) will be returned as zero.

`ircvtr` returns the value of its single precision IEEE mode input in native mode. A +/- zero (dirty, or true) will be returned as zero.

`idcvtd` returns the value of its double precision IEEE mode input in native mode. A +/- zero (dirty, or true) will be returned as zero.

Diagnostics

`rcvtir` and `dcvtid` have the following boundary conditions and return values:

Boundary condition	Return value
underflow	0
Rop	+Inf

`ircvtr` and `idcvtd` have the following boundary conditions and return values:

Boundary condition	Return value
+/- Inf	Rop
+- Nan	Rop
overflow	Rop

Files

```
/usr/lib/libm.a
/usr/lib/libF77.a
```

rename(3F)

Rename a file

Synopsis

```
integer function rename (from, to)
character*(*) from, to
```

Description

`from` must be the pathname of an existing file. `to` becomes the new pathname for the file. If `to` exists, then both `from` and `to` must be the same type of file, and must reside on the same filesystem. If `to` exists, it is removed first.

The returned value is 0 if successful; a system error code otherwise.

Files

`/usr/lib/libU77.a`

See Also

`rename(2), perror(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Notes

`rename` is an optional product; for more information, contact your CONVEX sales representative.

restart(3F)

Restart execution of a process or process family

Synopsis

```
integer function restart (path, flags, signo)
character*(*) path
integer flags
integer signo
```

Parameter	Meaning
path	The file name of the checkpoint file for the root of the process hierarchy.
flags	Optional actions to be performed by `restart`.
signo	Depending on the value of the `flags` argument, the `signo` parameter may be interpreted as an optional signal to send to restarted processes.

Description

The `restart` library routine restarts a process or process hierarchy from a checkpoint file.

By default the top level process of the process hierarchy will receive a SIGCONT signal. This may be modified with the `flags` field.

See `restart`(3) for a more complete description of the arguments to `restart`.

Return Value

`restart` returns the process id of the root level process if successful; otherwise an error code is returned.

Files

`/usr/lib/libc_old.a`

See Also

`chkpnt(3), restart(3), chkpnt(3f)`

rindex(3F), lnblnk

Tell about character objects

Synopsis

```
integer function rindex (string, substr)
character*(*) string, substr

function lnblnk (string)
character*(*) string
```

Description

`rindex` returns the index of the last occurrence of the substring `substr` in `string`, or zero if it does not occur. The intrinsic function, `index`, returns the index of the first occurrence of a substring. `lnblnk` returns the index of the last non-blank character in `string`. (Intrinsic function `len` returns the size of the character object argument.)

Files

`/usr/lib/libU77.a`

Notes

`rindex` is an optional product; for more information, contact your CONVEX sales representative.

secnds(3F)

Return time as a floating-point value

Synopsis

```
function secnds(time)
real time
```

Description

`secnds` returns the number of seconds since midnight less the value of its argument.

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2), date(3F), etime(3F), fdate(3F), idate(3F), stime(3F), time(3F)`

Notes

`secnds` is an optional product; for more information, contact your CONVEX sales representative.

setjmp(3F), longjmp, _setjmp, _longjmp

Non-local goto for FORTRAN

Synopsis

```
integer function setjmp(env)
integer*4 env(10)

subroutine longjmp(env, val)
integer*4 env(10),val

integer function _setjmp(env)
integer*4 env(10)

subroutine _longjmp(env, val)
integer*4 env(10),val
```

Description

These FORTRAN callable routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

`Setjmp` saves its stack environment in `env` for later use by

`longjmp`. It returns value 0. `Longjmp` restores the environment saved by the last call of `setjmp` . It then returns in such a way that execution continues as if the call of `setjmp` had just returned the value `val` to the function that invoked `setjmp`, which must not itself have returned in the interim. All accessible data have values as of the time `longjmp` was called.

`Setjmp` and `longjmp` save and restore the signal mask `sigmask (2)`, while `_setjmp` and `_longjmp` manipulate only the stack and registers.

`Longjmp` may not be used to restore the environment saved by `_setjmp`, and `_longjmp` may not be used to restore the environment saved by `setjmp`. An error condition occurs in both of these cases.

See Also

`sigvec(2), sigstack(2), signal(3C)`

signal(3F)

Change the action for a signal

Synopsis

```
integer function signal(signum, proc, flag)
integer signum, flag
external proc
```

Description

When a process incurs a signal (see `signal(3C)`), the default action is usually to clean up and abort. The user may choose to write an alternative signal-handling routine. A call to `signal` is the way this alternate action is specified to the system.

`signum` is the signal number (see `signal(3C)`). If `flag` is negative, then `proc` must be the name of the user signal-handling routine. If `flag` is zero or positive, then `proc` is ignored and the value of `flag` is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for `flag` have specific meanings: 0 means lquse the default actionrq (see "Notes" below); 1 means "ignore this signal."

A negative returned value is the previous action definition. The returned value can be used in subsequent calls to `signal` in order to restore a previous action definition. A positive returned value is a system error code. (See `perror(3F)`.)

Files

`/usr/lib/libU77.a`

See Also

`signal(3C), kill(3F), kill(1)`
CONVEX FORTRAN User's Guide, Chapter 8

Notes

`fc` arranges to trap certain signals when a process is started. The only way to restore the default `fc` action is to save the returned value from the first call to `signal` .

If the user signal-handling routine is called, it is passed the signal number as an integer argument.

`signal` is an optional product; for more information, contact your CONVEX sales representative.

sleep(3F)

Suspend execution for an interval

Synopsis

```
subroutine sleep (itime)
```

Description

`sleep` causes the calling process to be suspended for `itime` seconds. The actual time can be up to 1 second less than `itime` due to granularity in system timekeeping.

Files

`/usr/lib/libU77.a`

See Also

`sleep(3)`

Notes

`sleep` is an optional product; for more information, contact your CONVEX sales representative.

stat(3F), lstat, fstat

Get file status

Synopsis

```
integer function stat (name, statb)
character*(*) name
integer statb(12)

integer function lstat (name, statb)
character*(*) name
integer statb(12)

integer function fstat (lunit, statb)
integer statb(12)
```

Description

These routines return detailed information about a file. `stat` and `lstat` return information about file `name` ; `fstat` returns information about the file associated with FORTRAN logical unit `lunit`. The order and meaning of the information returned in array `statb` are as described for the structure `stat` under `stat(2)`. The "spare" values are not included.

The value of either function is zero if successful; an error code otherwise.

Files

`/usr/lib/libU77.a`

See Also

`stat(2), access(3F), perror(3F), time(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in `<sys/param.h>`.

Notes

`stat` is an optional product; for more information, contact your CONVEX sales representative.

stat(3F), lstat, fstat

stime(3F), ctime, ltime, gmtime

Return system time

Synopsis

```
integer function stime()

character*(*) function ctime (stime)
integer stime

subroutine ltime (stime, tarray)
integer stime, tarray(9)

subroutine gmtime (stime, tarray)
integer stime, tarray(9)
```

Description

`stime` returns the time measured in seconds since 00:00:00 GMT, Jan. 1, 1970. This is the value of the system clock.

`ctime` converts a system time to a 24-character ASCII string. The format is described under `ctime(3)`. No "newline" or NULL is included.

`Ltime` and `gmtime` disect time into month, day, etc., either for the local time zone or as GMT. The order and meaning of each element returned in `tarray` are described under `ctime(3)`.

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2), ctime(3), date(3F), etime(3F), fdate(3F), idate(3F), secnds(3F), time(3F)`

Notes

`stime` is an optional product; for more information, contact your CONVEX sales representative.

system(3F)

Execute a command

Synopsis

```
integer function system (string)
character*(*) string
```

Description

`system` causes `string` to be given to your shell as input as if the string had been typed as a command. If environment variable `SHELL` is found, its value is used as the command interpreter (shell); otherwise `sh(1)` is used.

The current process waits until the command terminates. The returned value is the exit status of the shell. See `wait(2)` for an explanation of this value.

Files

`/usr/lib/libU77.a`

See Also

`exec(2), wait(2), system(3)`

Diagnostics

Exit status 127 indicates the shell couldn't be executed.

Bugs

`String` cannot be longer than NCARGS-50 characters, as defined in `<sys/param.h>`.

Notes

`system` is an optional product; for more information, contact your CONVEX sales representative.

time(3F)

Return time in an ASCII string

Synopsis

```
subroutine time (buf)
character *(*) buf
```

Description

`time` returns the time as hh:mm:ss. `Buf` is any 8-byte variable or character string.

Files

`/usr/lib/libU77.a`

See Also

`gettimeofday(2), ctime(3), date(3F), etime(3F), fdate(3F), idate(3F), secnds(3F), stime(3F)`

Notes

`time` is an optional product; for more information, contact your CONVEX sales representative.

topen(3F), topenreadonly, tclose, tread, tretry, twrite, trewind, tweof, terase, tskipf, tskipr, toffl, tstate, taset, tawait

FORTRAN tape I/O

Synopsis

```
integer function topen (tlu, devnam)
integer tlu
character*(*) devnam

integer function topenreadonly (tlu, devnam)
integer tlu
character*(*) devnam

integer function tclose (tlu)
integer tlu

integer function tread (tlu, buffer, len)
integer tlu, len
character*(*) buffer

integer function twrite (tlu, buffer, len)
integer tlu, len
character*(*) buffer

integer function tretry (tlu, flag)
integer tlu
logical flag

integer function trewind (tlu)
integer tlu

integer function tweof (tlu)
integer tlu

integer function terase (tlu)
integer tlu

integer function tskipf (tlu, nfiles)
integer tlu, nfiles

integer function tskipr (tlu, nrecs)
integer tlu, nrecs
```

```
integer function toffl (tlu)
integer tlu

integer function tstate (tlu, fileno, errf, eoff,
          eotf, type, dsreg, erreg, resid, size)
integer tlu, fileno, dsreg, erreg, resid, size
logical errf, eoff, eotf

integer function taset (tlu, tflags)
integer tlu, tflags

integer function tawait (tlu, count)
integer tlu, count
```

Description

These functions provide a low-level interface between FORTRAN and magnetic tape devices. They apply only to "raw" mode tape files (rmt0, ...,rmt23). See `mtio(4)` for details on the raw interface to magnetic tape. A tape logical unit (`tlu`) is `topen`ed in much the same way as a normal FORTRAN logical unit is `open`ed. All other operations are performed via the `tlu`. The `tlu` has no relationship at all to any normal FORTRAN logical unit.

`topen` associates a "raw" mode device name with a `tlu`. The device name must be a string of length 21 or less. `tlu` must be in the range 0 to 19. `topen` closes the file, if it is currently open. The normal return value is 0. If the value of the function is negative, an error has occurred.

`topenreadonly` opens the tape with readonly mode. There is no enforcement of the presence of a write ring. The device name must be a string of length 21 or less.

`tclose` closes the tape device and removes its association with `tlu`. The normal return value is 0. A negative value indicates an error.

`tread` reads the next physical record from tape to `buffer`. The `buffer` need not be of type character—any data type is acceptable. `len` is the number of bytes to be read. The return value is: 1) the number of bytes read (synchronous access), or 2) the number of bytes requested for read (asynchronous access). If the value is 0, then end-of-file or end-of-tape has been detected. A negative value indicates an error.

`twrite` writes a physical record to tape from `buffer`. The `buffer` need not be of type character—any data type is acceptable. `len` is the number of bytes to be written. The return value is: 1) the number of bytes

written (synchronous access), or 2) the number of bytes requested to write (asynchronous access). A nonpositive return value indicates an error.

`tretry` enables or disables retry on tape errors. If `flag` is `true,` then the driver attempts to recover from tape errors. If `flag` is `false,` then no attempt is made to recover from tape errors. Once `tretry` has been called, the retry value set via `flag` applies to all subsequent `treads` and `twrites,` until another call to `tretry` is made. By default, retry on tape errors is enabled. The normal return value is 0. A negative value indicates an error.

`trewind` rewinds the tape associated with `tlu` to the beginning of the first data file. The normal return value is 0. A negative value indicates an error.

`tweof` writes an end-of-file record to tape. The normal return value is 0. A negative value returned indicates an error.

`terase` writes an interrecord gap to tape. The normal return value is 0. A negative value returned indicates an error.

`tskipf` repositions the tape file either forward or backward, depending on the value of `nfiles.` `nfiles` is a relative file position. If `nfiles` is positive, then `nfiles` end-of-file marks are skipped `forward.` If `nfiles` is negative, then `nfiles` end-of-file marks are skipped `backward.` The normal return value is 0. A negative value indicates an error.

`tskipr` repositions the tape file either forward or backward, depending on the value of `nrecs.` `nrecs` is a relative physical record position. If `nrecs` is positive, then `nrecs` interrecord gaps are skipped `forward.` If `nrecs` is negative, then `nrecs` interrecord gaps are skipped `backward.` The normal return value is 0. A negative value indicates an error.

`toffl` rewinds the tape and puts the drive offline. The normal return value is 0. A negative function value indicates an error.

`tstate` allows the user to determine the state of the tape file. `fileno` is set to the current file number (0 being the first file). The logical values `errf, eoff,` and `eotf` indicate an error has occurred, the current file is at EOF, or the tape has reached logical end-of-tape. End-of-tape (EOT) is indicated by an empty file (i.e., two adjacent EOF marks). It is not allowable to read past EOT, although it is allowable to write. The values of `type, dsreg, erreg, resid,` and `size` are set to the corresponding values returned by the mag tape status command. These

indicate the current physical state of the tape device. `type` is the type of the magtape device, `dsreg` is the device independent status of the tape channel, `erreg` is the drive error register, `resid` is the residual count, and `size` is the maximum allowable record size. See `mtio(4)` for details.

`taset` allows the user to set/reset the asynchronous status of the tape. `tflags` are:

tflag	Meaning
A_ASYNC '1'X	Enables asynchronous IO to the specified unit.
A_FNCACHE '2'X	Bypasses the system buffer cache when possible.
A_SINGLE '4'X	Allows only a single pending IO request to be active at once for the specified unit. An implied `tawait` is performed before each `tread` or `twrite`; the results are discarded and errors are returned immediately.
A_SIGNAL '8'X	Generates a SIGIO signal on completion of all pending asynchronous I/O operations for the specified unit. The signal can be caught by a user-specified FORTRAN subroutine; see `signal`(3F) for details.

`tawait` waits for the completion of all pending I/O on the specified unit and returns the SUM count of bytes transferred since the last call to `tawait` for the specified unit. When an error occurs, `tawait` returns the error code of the first error detected.

Diagnostics

All of the FORTRAN tape functions return a negative value on an error condition. Except for `tread` and `twrite`, a zero function value indicates that the operation terminated normally. `tread` and `twrite` return the number of bytes actually transferred. If an error does occur during any tape operation, the global variable `errno` is set to the error number, which may be either a system error (see `intro`(2)) or a FORTRAN I/O error (see `perror`(3F)). The value of this variable is accessible via the FORTRAN utility `perror`.

Files

`/usr/lib/libU77.a`

See Also

`intro(2), mtio(4), perror(3F), signal(3F)`

Notes

`topen` is an optional product; for more information, contact your CONVEX sales representative.

traper(3F)

Trap arithmetic errors

Synopsis

```
integer function traper (mask)
```

Description

By default, floating-point underflow, integer overflow, and intrinsic errors are not trapped during execution. This routine changes the setting of the floating-point underflow and integer overflow traps by setting or resetting the status bits in the process status word.

The trap-enable bits are inherited from the calling subprogram, and the previous state is restored on return. If integer overflow or floating-point underflow occurs and trapping is enabled, signal SIGFPE is sent to the process. (See `signal (3C)`.)

By default, the occurrence of an intrinsic error (e.g., square root of a negative real value) does not terminate execution of the program. Setting the intrinsic error trap causes the program to terminate on the first intrinsic error encountered. The intrinsic error trap value applies to the entire program.

The argument to `traper` is produced by summing the appropriate flags below:

Value	Meaning
'1'X	trap integer overflow
'2'X	trap floating underflow
'4'X	trap intrinsic errors

The previous value of these flags is returned.

Files

`/usr/lib/libU77.a`
CONVEX FORTRAN User's Guide, Chapter 8

See Also

`signal(3C), signal(3F)`

Notes

`traper` is an optional product; for more information, contact your CONVEX sales representative.

ttynam(3F), isatty

Find name of a terminal port

Synopsis

```
character*(*) function ttynam (lunit)

logical function isatty (lunit)
```

Description

`ttynam` returns a blank padded path name of the terminal device associated with logical unit `lunit`.

`isatty` returns .TRUE. if `lunit` is associated with a terminal device; .FALSE. otherwise.

Files

`/dev/*`
`/usr/lib/libU77.a`

Diagnostics

`ttynam` returns an empty string (all blanks) if `lunit` is not associated with a terminal device in directory `/dev`.

Notes

`ttynam` is an optional product; for more information, contact your CONVEX sales representative.

unlink(3F)

Remove a directory entry

Synopsis

```
integer function unlink (name)
character*(*) name
```

Description

`unlink` causes the directory entry specified by pathname `name` to be removed. If this was the last link to the file, the contents of the file are lost. If `unlink` is successful it returns a value of zero. If not, a system error code is returned.

Files

`/usr/lib/libU77.a`

See Also

`unlink(2), link(3F), filsys(5), perror(3F)`

Bugs

Pathnames can be no longer than MAXPATHLEN as defined in `<sys/param.h>`.

Notes

`unlink` is an optional product; for more information, contact your CONVEX sales representative.

wait(3F)

Wait for a process to terminate

Synopsis

```
integer function wait (status)
integer status
```

Description

`wait` causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last `wait`, return is immediate; if there are no children, return is immediate with an error code.

If the returned value is positive, it is the process ID of the child and `status` is its termination status (see `wait(2)`). If the returned value is negative, it is the negation of a system error code.

Files

`/usr/lib/libU77.a`

See Also

`wait(2), signal(3F), kill(3F), perror(3F)`

Notes

`wait` is an optional product; for more information, contact your CONVEX sales representative.

Index

!

statement 352
#include statement 163
#undef statement 378
$ descriptor 323
%LOC function 79, 346
%REF function 78, 345
%VAL function 77, 345
-72 option 19, 160
-a1 option 12
-ansic 18
-B option 18
-c option 9
-cfc option 5, 409
-cs option 12
-D preprocessor option 17, 379
-db option 13
-dc option 13, 159
-dfc option 5, 289
-ds option 6
-E 17
-E preprocessor option 379
-ep option 6
-F66 option 5
-f90 option 6
-fi option 9
-fn option 9
-fpp preprocessor option 17, 379
-i option 10
-I preprocessor option 17, 379
-i2 option 10
-i4 option 10
-i8 option 10
-il option 7, 28
-is option 7, 28
-iw option 15–16
-link 18
-LST option 14
-LSTI option 14
-na option 14
-no option 7
-nopeel 7
-noptst 7
-nosc option 18, 238
-nv option 15
-nw option 15
-O option 7, 18
-O0 option 26
-O1 option 7, 27
-O2 option 7, 27
-O3 optimization 27
-O3 option 7
-or option 15
-p option 13, 70
-p8 option 10
-pa option 14
-pab option 14
-par 14
-pb option 13, 70
-pcc option 18
-pd8 option 10
-peel option 8
-peelall 8
-pg option 13, 70
-pp 17
-ptst 8
-ptstall 8
-pw option 16
-r option 11
-r4 option 11
-r8 option 11
-re option 11
-rl option 8, 28
-S option 11
-sa option 6
-sc option 14
-sfc option 6, 425
-sl option 16
-tl option 18
-tm option 12
-U preprocessor option 17, 380
-uo option 9
-ur option 9
-vfc option 6, 417
-vn option 19
-xr option 15
-xr1 option 16
-xra option 15
-xrf option 15
-xrg option 15
-xro option 16
-xrr 15
: descriptor 324

A

abort man page 469–470
absolute value 140
ACCEPT statement 246, 260
access
  direct 37
  sequential 37
ACCESS keyword 273
access man page 471–472

access modes 37
accessing files 249
actual arguments 337
adb 66
adjustable arrays 339
alarm man page 473
ALL intrinsic 186
Allocatable arrays 215
ALLOCATABLE statement 215
alternate return arguments 343, 348
analyzer
  performance 14
AND
  bitwise 137
ANSI Fortran 90 standard 184
ANSI standard formatting 160
ANSI-standard formatting 160
ANY intrinsic 186
apostrophe descriptor 305
arc cosine 130
arc sine 129–130
arc tangent 130
  two arguments 131
  two arguments (degree) 131
arc tangent (degree) 131
aread man page 475
argument packets 75, 83
argument passing examples 83
argument pointer 75
argument-passing mechanisms 77
arguments 125, 348
  actual 350
  alternate return 348
  character 341
  dummy 348, 351
  hollerith 341
arithmetic expression
  data types 194
arithmetic expressions 194
arithmetic IF statement 232
arithmetic operators 145, 194
array arguments 339
array construction and manipulation functions 190
array declaration 176
array intrinsics
  Fortran 90 184
array reduction functions 185
array storage 184
array subscripts 181
array table 22
arrays 175
  automatic 411
  conformability 176
  rank 176
  shape 176
ASCII character set 403
aset man page 475
assembly-language debugger 66
ASSIGN statement 227
ASSIGN_LOCK directive 386
assigned GOTO statement 231
assignment statements 223
ASSOCIATEVARIABLE keyword 274
assumed-length character argument 206
assumed-size arrays 340
asterisk descriptor 305
  runtime formats 327
automatic arrays 411
auxiliary I/O operations 150
auxiliary input/output statements 271
await man page 475
awrite man page 475

B

B descriptors 316
BACKSPACE statement 286–287
basic block profiler 70
BEGIN_ORDER directive 387
BEGIN_SECTION directive 387
BEGIN_TASKS directive 388
besj0 man page 479
besj1 man page 479
besjn man page 479
bessel functions 479
besy0 man page 479
besy1 man page 479
besyn man page 479
Binary Data File Format Conversions 288
bitwise AND 137
bitwise circular shift 139
bitwise clear 139
bitwise complement 137
bitwise extract 138
bitwise OR 137
bitwise set 138
bitwise shift 138
bitwise test 139
bitwise XOR 137
blank common storage 202
BLANK keyword 274, 407
BLOCK DATA statement 357
BLOCK DATA statement in subprograms 350
block data subprogram 357
  with common block 358
block IF statement 234
block-level profiling 68
BLOCKSIZE keyword 274
BN descriptor 316
bprof profiler 13, 70
built-in functions 345
BYTE 112, 166
BZ descriptor 316

C

C interface 24
CALL statement 243, 350–351
calling conventions 75, 125
calling utility routines 89
carriage-control characters 335
CARRIAGECONTROL keyword 275
CHAR function 48
character arguments 341
character constants 43, 174
character conversions 224
character data 43
character descriptor A 303
character equivalence 209
character expressions 198
CHARACTER FUNCTION statement 349
character library functions 47
character relationals 140
character representation 116
character set 155–156
    extended 156
    standard 155
character strings
    concatenating 46
character substrings 45, 198
CHARACTER type-declaration statements 205
character variables
    declaring 44
character-per-column formatting 158
character-valued function 76
characters
    carriage-control 335
chdir man page 481
chkpnt man page 483
chmod man page 485
circular shift
    bitwise 139
clear
    bitwise 139
CLOSE statement 282
code-generation options 9
colon descriptor 324
comma field separator 326
comment line 156
common block
    in block data subprogram 358
common block storage 202
common blocks 209
    Cray TASK COMMON 415
common logarithm 128
COMMON statement 175, 202
commonly used library routines 374
comparison operators 145
compiler directives 160, 383
compiler features 3
compiler messages 20, 117
compiler options 5
compiling FORTRAN 66 programs 405
compiling programs 5
complement
    bitwise 137
COMPLEX 166
    imaginary part 133
    real part 133
    changing default size 10
complex conjugate 133
complex descriptor 303
complex programmed operators 144
complex representation 116
complex values 125
COMPLEX*16 166
COMPLEX*16 constants 170
COMPLEX*8 166, 195–196
COMPLEX*8 constants 170
COMPLEX-to-COMPLEX conversions 168
COMPLEX-to-noncomplex conversions 168
computed GOTO statement 230
concatenation 198
conditional evaluation
    short circuiting 237
conformability
    defined 176
    scalar value 176
conjugate
    complex 133
constant expressions 199
constants 168
    Cray octal 415
    default sizes 9
    translation of REAL 9
consultant package 69
continuation indicator 159
continuation line 159
CONTINUE statement 243
control statement 229
conversion
    fix-to-float 133
    float-to-fix 132
    integer 141
    REAL*16 to REAL*4 142
    REAL*16 to REAL*8 142
    REAL*4 to REAL*16 142
    REAL*8 to REAL*16 142
    user-defined 292
    automatic 167
conversion feature
    when to use 289
conversion of data types 167
conversion restrictions 290
Conversion Using a Shell Variable 296
Conversion Using OPEN Statement 289
conversions
    IEEE/native 140
CONVEX consultant 69
CONVEX FORTRAN 3
CONVEX math library 125
CONVEX symbolic debugger 69
ConvexOs Utilities 90
cosine 129

hyperbolic 131
COUNT 187
COVUEshell 163, 271
Cray automatic arrays 411
Cray compatibility 409
Cray constants
octal 415
Cray data files 39
Cray intrinsics 414
cray library routines 414
Cray support
asterisk descriptor 305, 327
TASK COMMON 415
cross-reference generator 15, 51
old 16
csd 69
ctime man page 573
CXdb 71
CXpa 14, 67

D

D descriptor 312
D indicator 159
Data File Formats 39
data files
Cray 39
data format 248, 297
data representation 111
data representations 81
DATA statement 217
data types 165
arithmetic expression 194
conversion 167
equivalenced 207
default sizes 9
data-type length specifiers 205
dataformat 279
date man page 487, 499
date routine 374
date utility 93
dbesj0 man page 479
dbesj1 man page 479
dbesjn man page 479
dbesy0 man page 479
dbesy1 man page 479
dbesyn man page 479
dcvtid man page 555
debug statements 159
debugger
assembly-language 66
CONVEX CXdb 71
symbolic 69
debugging options 12
DECODE statement 268
default data types 9
default descriptor values 325
DEFAULTFILE keyword 276
derf man page 489
derfc man page 489
descriptors 303
development tools
program 51
diagnostic messages 20
difference
positive 135
dimension declarator 176
DIMENSION statement 206
direct access 37, 249
direct-access file 38
direct-access WRITE statements 264
directives
compiler 160, 383
DISPOSE keyword 277
DO list
implied 219
DO loops
extended range 240
DO statement 238, 240, 242
DO WHILE statement 241–242
dollar sign descriptor 323
DOT_PRODUCT intrinsic 184
DOUBLE PRECISION 166
double precision constants 169
drand man page 553
dummy argument 337
NAMELIST statement 212
dummy arguments 338, 348
arrays 339
dump
postmortem 70

E

E descriptor 312
edit descriptors 303
ENCODE statement 266
END DO statement 242
END specifier 98
END statement 244
END statement in subprograms 350
end-of-file specifier 254
END_ORDER directive 387
END_SECTION directive 387
END_TASKS directive 388
ENDFILE record 247
ENDFILE statement 286
entry points
I/O list element transmission 149
I/O list initialization 148
intrinsics 126
EQUIVALENCE statement 175, 206
equivalencing
arrays 207
erf man page 489
erfc man page 489
ERR keyword 277
ERR specifier 98

error function man pages 489
error messages 20
error specifier 254
error utility 73, 117
error-processing utilities 102
errors
  runtime 97
errsns man page 491
errsns routine 374
errsns utility 94
errtrap man page 493
errtrap utility 102
etime man page 495
evaluation
  short circuit 237
exception
  runtime 97
exceptions 101
executable program 155
executing programs 19
exit man page 497
exit routine 374
exit utility 94
exponential 128
exponentiation programmed operators 143
expressions 194
  constant 199
external files 36
external READ statements 256
EXTERNAL statement 213
extract
  bitwise 138

F

F descriptor 310
fc command line 5
fc man page 431
fct man page 449
fcunblock man page 447
fcxref man page 451
fcxref program 15, 51
fgetc man page 511
field declaration 422
field descriptors 303
field separators
  external 326
FILE keyword 277
file pointers 39
file positioning 302
file type 36
file-naming conventions 4
file-positioning statements 286
files 247
  FORTRAN source 4
  accessing 249
FIND statement 270
fix-to-float conversion 133
flmin man page 501
float-to-fix conversion 132
floating-point
  IEEE 115
  native 114
floating-point data representation 112
floating-point representation
  IEEE 3, 9
flush man page 503
for$ prefix 126
for$getfp 39
FORCE_PARALLEL directive 390
FORCE_PARALLEL_EXT directive 389
FORCE_VECTOR directive 390
fork man page 505
FORM keyword 278
format code separators 408
FORMAT control 301
format conversions 288
format specifications 299
format specifier 252
FORMAT statement 299
  apostrophe descriptor 305
  asterisk descriptor 305
formats
  variable 327
formatted I/O 36
formatted records 246
formatting
  ANSI 160
  list-directed 328
FORTRAN 66 compatibility 405
FORTRAN 77 formatting 160
Fortran 90
  ANSI standard 184
Fortran 90 intrinsics 142, 184, 374
  ALL 186
  ANY 186
  array location functions 193
  construction and manipulation 190
  COUNT 187
  DOT_PRODUCT 184
  MATMUL 185
  MAXLOC 193
  MAXVAL 187
  MERGE 190
  MINLOC 193
  MINVAL 188
  optional arguments 184
  PACK 190
  PRODUCT 188
  reduction functions 185
  SPREAD 191
  SUM 189
  TRANSPOSE 192
  UNPACK 192
  vector and matrix multiply 184
FORTRAN argument packets 75
FORTRAN character set 403
FORTRAN I/O library 147
FORTRAN intrinsic library 125

fpp 163, 377
fpr man page 455
fputc man page 549
FREE_LOCK directive 386
fseek man page 507
fsplit man page 457
fsplit utility 19
fstat man page 571
ftell man page 507
FUNCTION statement 349
function subprograms 348
function-naming convention 125
functions 344
  intrinsic 127
fxref man page 459
fxref program 16

G

G descriptor 314
generic and specific intrinsics 359
  table 359
gerror man page 545
gerror utility 107
getarg man page 509
getc man page 511
getcwd man page 513
getenv man page 515
getgid man page 521
getlog man page 517
getpid man page 519
getuid man page 521
global scalar optimization 27
gmtime man page 573
GOTO statement 229
gprof profiler 13, 70
graph profiler 70

H

H descriptor 306
hardware intrinsics 104
hexadecimal constants 171
Hollerith constants 173
  Cray 416
hollerith constants as arguments 341
Hollerith representation 116
hostnm man page 523
hyperbolic cosine 131
hyperbolic sine 131
hyperbolic tangent 131

I

I descriptor 307
I/O error processing 97
I/O forms 36
I/O list element transmission 149
I/O list initialization 148
I/O list termination 149
I/O operation 147
I/O runtime routine naming convention 148
I/O statement format 250
I/O statements 245
iargc man page 509
ICHAR function 47
idate man page 525
idate routine 374
idate utility 93
idcvtd man page 555
IEEE 754 standard 151
IEEE compatibility 151
IEEE floating-point 115
IEEE floating-point representation 3, 9
IEEE/native conversions 140
ierrno man page 545
ierrno utility 107
IF statement
  short circuiting 237
IF statements 232
IF THEN statement 234
if-do optimizations 29
imaginary part of complex 133
IMPLICIT NONE statement 203
IMPLICIT statement 203
implied-DO list 219, 251
INCLUDE statement 163, 417
index
  string 140
INDEX function 49
Inf operand 115, 312, 314, 316
initial line 159
inline substitution 28
input
  list-directed 329
input/output 31
input/output lists 250
input/output statements 245
  auxiliary 271
INQUIRE statement 283
INTEGER
  nearest 141
  changing default size 10
integer constants 169
integer conversion 141
integer descriptor 303
integer part of real 133
integer representation 112
INTEGER*1 166
INTEGER*2 166
INTEGER*4 166
INTEGER*8 166

INTEGER-to-REAL conversions 168
internal files 36, 38, 247
internal READ statements 259
internal WRITE statements 265
intrinsic functions 127, 344, 359
intrinsic library 125
INTRINSIC statement 213
intrinsics
  Cray 414
  Fortran 90 184, 374
  generic and specific 359
intro man page 463
intro(1F) man page 429
invoking the compiler 5
ioinit man page 527
IOSTAT keyword 279
IOSTAT specifier 98
irand man page 553
ircvtr man page 555
itime man page 525

K

keywords
  OPEN statement 271
kill man page 531

L

L descriptor 306
LEN function 48
length
  string 140
lexical comparison functions 49
libc.a 18
libcfc man page 533
libcfc.a 412
libF77 library 125
libF90 man page 537
libI77.a library 147
libraries
  runtime 124
library
  intrinsic 125
  math 125
library routines 359, 374
libU77.a 123
  Cray support 414
limits
  system 401
link man page 539
list
  implied-DO 219
list-directed
  null value 330
  slashes 330
list-directed character input 329
list-directed complex input 329
list-directed formatting 328
list-directed I/O 36
list-directed input 329
list-directed output 333
listing options 14
LNBLNK function 48
lnblnk man page 561
loader 19
loading
  suppressing 9
loading programs 19
loc man page 541
local scalar optimization 26
locatoin functions 193
logarithm
  common 128
  natural 127
LOGICAL
  changing default size 10
logical constants 174
logical descriptor 303
logical elements 197
logical entities 196
logical expressions 197
logical IF statement 233
logical names 32
logical operator .XOR. 198
logical records 38
logical representation 111
LOGICAL*1 166
LOGICAL*2 166, 195
LOGICAL*4 166
LOGICAL*8 166
longjmp man page 565
longjmp utility 102
loop replication 8, 28
loop table 20
loop-level profiling 68
lstat man page 571
ltime man page 573

M

machine-dependent optimization 26
main program 155, 211
MAP keyword 423
math library 125
MATMUL intrinsic 185
matrix
  defined 184
MAX_TRIPS directive 391
maximum 134
MAXLOC 193
MAXREC keyword 279
MAXVAL 187
MERGE 190
message options 14
messages 20, 117

minimum 134
MINLOC 193
MINVAL 188
miscellaneous compiler options 17
mth$ prefix 126
multiple statements 159
mvbits man page 543
mvbits routine 375
mvbits utility 95

N

NAME keyword 277
namelist input 330
namelist specifier 255
NAMELIST statement 212
namelist-directed formatting 330
namelist-directed I/O 38
namelist-directed output 334
names
  logical 32
  runtime routine 125
NaN operand 115, 312, 314, 316
native floating-point 114
natural logarithm 127
nearest integer 141
nested block IF statement 237
nested DO loops 240
NEXT_TASK directive 388
NML keyword 255
NO_PARALLEL directive 391
NO_PEEL directive 391
NO_PROMOTE_ALL directive 391
NO_RECURRENCE directive 392
NO_SIDE_EFFECTS directive 392
NO_VECTOR directive 393
non-FORTRAN-to-FORTRAN calling sequence 79
noncomplex-to-COMPLEX conversions 168
nonrepeatable descriptors 301
NOSPANBLOCKS keyword 280
numeric type-declaration statements 204

O

O descriptor 308
octal constants 170, 415
old cross-reference generator 16
OPEN statement 33, 271
OPEN statement conversions 289
OPEN statement keywords 407
operator precedence 195
optimization 25
  global scalar 27
  local scalar 26
  machine-dependent 26
optimization report 15, 20
optimizations
  if-do 29
optional arguments to Fortran 90 intrinsics 184
options
  compiler 5
  -nosc 238
  miscellaneous 17
  preprocessor 379
OPTIONS statement 161
OR
  bitwise 137

P

P descriptor 318
PACK 190
packets
  argument 75
parallelization 27
PARAMETER statement 210
  alternate 210
PAUSE statement 244
PEEL directive 393
PEEL_ALL directive 393
performance analyzer 14, 67
perror man page 545
perror utility 107
pmd utility 70
pointer
  argument 75
positive difference 135
postmortem dump 70
precedence
  operator 195
preconnection of units 32
PREFER_PARALLEL directive 394
PREFER_PARALLEL_EXT directive 394
PREFER_VECTOR directive 394
preprocessor 163, 377
PRINT statement 246, 266
priority
  data type 195
procedure names 80
procedures as dummy arguments 343
PRODUCT 188
prof profiler 13, 70
profiler
  bprof 13
  CXpa 14
  gprof 13
  prof 13
profilers 70
profiling options 12
program
  executable 155
  main 155, 211
program development tools 51
program interfaces 24
PROGRAM statement 155, 211
PROGRAM statement in subprograms 350

program units 155
PROMOTE TEST directive 394
PROMOTE_TEST_ALL directive 394
PSTRIP directive 395
putc man page 549

Q

Q descriptor 323
qrand man page 553
qsort man page 551

R

R descriptor 320
ran man page 553
ran routine 374
ran utility 94
rand man page 553
rank 176
    data type 195
rcvtir man page 555
READ statement 246, 255
    direct 258
    external 256, 258
    internal 259
    sequential 256
READONLY keyword 280
REAL 166
    changing default size 11
    integer part 133
    changing default size 10
real constants 169
    translation 9
real data representation 112
real descriptor 303
real part of complex 133
REAL*16 125, 169, 195
REAL*16 data representation 112
REAL*16 programmed operators: 144
REAL*16 to REAL*4 conversion 142
REAL*16 to REAL*8 conversion 142
REAL*4 166, 169, 225
REAL*4 data representation 112
REAL*4 to REAL*16 conversion 142
REAL*4 to REAL*8 conversion 141
REAL*8 166, 169, 195–196, 225
REAL*8 data representation 112
REAL*8 product of F102 134
REAL*8 to REAL*16 conversion 142
REAL*8 to REAL*4 conversion 142
REAL*8-to-REAL*4 conversions 168
REAL-to-INTEGER conversions 168
REAL-to-REAL conversions 168
RECL keyword 280
RECORD keyword 421
record specifier 253
records 246
    logical 38
RECORDSIZE keyword 280
RECORDTYPE keyword 280
reentrant code
    generating 11
registers
    scalar 125
    vector 125
relationals
    character 140
remainder 136
rename man page 557
repeatable descriptors 301, 303
replication
    loop 8
report
    optimization 15, 20
reserved operand 114, 312, 314, 316
restart man page 559
Restrictions on Conversions 290
RETURN
    alternate 351
RETURN statement 243–244, 351, 353
RETURN statement in subprograms 350
return values 81
REWIND statement 286–287
RINDEX function 49
rindex man page 561
Rop operand 114, 312, 314, 316
routine-level profiling 67
ROW_WISE directive 395
runtime error messages 23, 118
runtime errors and exceptions 97
runtime formats 326
    asterisk descriptor 327
runtime interface 24
runtime libraries 123–124
runtime messages 117
runtime routine data items 146
runtime routine names 125
runtime routine prefixes 126
runtime stack 79
runtime utilities 89

S

S descriptor 319
SAVE statement 214
SCALAR directive 396
scalar registers 125
scalar truncation 125
scalar value conformability 176
scale factor 318
secnds man page 563
secnds routine 374
secnds utility 94
SELECT directive 397
semicolon separator 159

sequential access 37, 249
sequential READ statements 256
sequential WRITE statement
  unformatted 263
sequential-access file 38
sequential-access WRITE statements 262
set
  bitwise 138
setjmp man page 565
setjmp utility 102
shape 176
shell variable conversions 296
shell variables 248
shift
  bitwise 138
  bitwise circular 139
short-circuit evaluation 237
sign
  transfer of 137
signal handling examples 107
signal man page 567
signal utility 106
signals and exceptions 99
sine 128
  hyperbolic 131
slash descriptor 324
sleep man page 569
source files 4
SP descriptor 319
specification statements 201
specifiers 252
SPREAD 191
SS descriptor 319
stack
  runtime 79
standard profiler 70
stat man page 571
statement field 159
statement function reference 348
statement functions 347
statement label 158
statement label assignment 227
statements
  executable 157
  multiple 159
  nonexecutable 157
STATUS keyword 281, 408
status specifier 254
stime man page 573
STOP statement 243
storage
  array 184
string index 140
string length 140
string-manipulation programmed operators 146
strip mining
  parallel 395
  vector 400
structure declaration 421
STRUCTURE keyword 421
SU descriptor 320
subprogram 155
subprogram calling conventions 75
subprograms 337
  block data 357
SUBROUTINE statement in subprograms 350
subroutine subprograms 350
SUM 189
Sun FORTRAN compatibility 425
symbolic debugger 69
symbolic Names 165
SYNCH_PARALLEL directive 398
system errors 118
system limits 401
system man page 575
system utilities 89
system utility 92

T

T descriptors 321
tab character 160
tab-key formatting 160
table
  array 22
tangent 129
  hyperbolic 131
target machine 12
taset man page 579
TASK COMMON statement 415
tasking directives 388
tawait man page 579
tclose man page 579
terase man page 579
test
  bitwise 139
time man page 577
time routine 374
time utility 94
TL descriptor 321
toffl man page 579
tools
  program development 51
topen man page 579
topenreadonly man page 579
TR descriptor 321
traceback utility 106
transfer of sign 137
TRANSPOSE 192
traper man page 585
tread man page 579
tretry man page 579
trewind man page 579
tskipf man page 579
tskipr man page 579
tstate man page 579
ttynam man page 587
tweof man page 579
twrite man page 579

TYPE keyword 281
TYPE statement 246, 266
type-declaration statements 204

U

unconditional GOTO statement 229
unformatted I/O 36
unformatted records 246
UNION keyword 422
UNIT keyword 282
unit specifier 252
units 247
    input/output 31
    redirection 32
    program 155
unlink man page 589
UNPACK 192
UNROLL directive 399
User-Defined Conversions 292
utilities 89
    ConvexOS 90
utility routines
    how to call 89

V

values
    complex 125
variable formats 327
variables 175
VAX FORTRAN compatibility 417
VAX FORTRAN records 420
VAX-11 utilities 93
vector
    defined 184
vector and matrix multiply functions 184
vector mask programmed operators: 145
vector registers 125
vectorization 27
VMS FORTRAN compatibility 417
VSTRIP directive 399

W

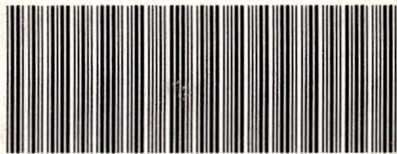
wait man page 591
WRITE statement 246, 261, 333
    list-directed output formats 333
WRITE statements
    direct access 264
    internal 265

X

X descriptor 321, 408
X3.198-199x ANSI standard 184
XOR
    bitwise 137

Z

Z descriptor 309



Order Number
DSW-038

Document Number
720-000130-101