FORTRAN 3.0.1 User's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-7250-10
Revision A, August 1994

Contents

Preface xxi

1. Introduction 1

1.1 Operating Environments 2

1.2 Standards 3

1.3 Extensions 3

1.4 New Features and Behavior Changes 5

New Features in 3.0.1 5

New Features in 3.0 6

Differences for FORTRAN in Solaris 2.x / 1.x / x86 7

Behavior Changes 8

1.5 Compatibility 13

1.6 Text Editing 14

1.7 Program Development 15

1.8 Debugging 15

1.9 Licensing 16

2. Getting Started 17
2.1 Summary 17
2.2 Compiling 18
2.3 Running 18
2.4 Renaming the Executables 18
3. Using the Compiler 21
3.1 Compile Command 22
Compile Link Sequence 22
Consistent Compile and Link 23
3.2 Language Preprocessor 23
3.3 Command-line File Names 24
3.4 Unrecognized Arguments 24
3.5 Compiler Options 25
Actions / Options Frequently Used 25
Actions Summary (Actions / Options Sorted by Action) 26
Options Details (Options / Actions Sorted by Option) 29
3.6 Native Language Support 63
Locale 63
Compile-Time Error Messages 64
3.7 Compiler Directives 67
General Pragmas 67
Parallel Pragmas 68

3.8 Miscellaneous Tips 69
Floating-Point Hardware Type 69
Many Options on Short Commands. 69
Align Block 70
Optimizer Out of Memory. 71
BCP Mode: How to Make 1.x Applications Under 2.x. 74
4. File System and FORTRAN I/O 77
4.1 Summary 77
4.2 Directories 79
4.3 File Names. 79
4.4 Path Names 79
Relative Path Names 80
Absolute Path Names. 80
4.5 Redirection 82
4.6 Piping. 83
5. Disk and Tape Files 85
5.1 Accessing Files from FORTRAN Programs. 85
Accessing Named Files 85
Accessing Unnamed Files 88
Passing File Names to Programs 88
Direct I/O 92
Internal Files 93

5.2 Tape I/O 95
Using TOPEN for Tape I/O 95
FORTRAN Formatted I/O for Tape 96
FORTRAN Unformatted I/O for Tape 96
Tape File Representation 96
End-of-File 97
Access on Multiple-file Tapes 97
6. Program Development 99
6.1 Simple Program Builds 99
Writing a Script 99
Creating an Alias 100
Using a Script or Alias 100
Limitations 100
6.2 Program Builds with the `make` Program 100
The `makefile` 100
Using `make` 102
The C Preprocessor 102
Macros with `make` 103
Overriding Macro Values 104
Suffix Rules in `make` 105
6.3 Tracking and Controlling Changes with SCCS 105
Putting Files under SCCS 106
Checking Files Out and In 111

7. Creating and Using Libraries 113

7.1 Libraries in General 113

Advantages of Libraries. 114

Disadvantages of Libraries 114

Debug Aids 115

Consistent Compile and Link 116

Fast Directory Cache for the Link-editor 116

7.2 Library Search Paths and Order 117

Order of Paths Critical for Compile (Solaris 1.x) 117

Library not Found. 118

Search Order for Library Search Paths. 120

7.3 Static Libraries 122

7.4 Dynamic Libraries 125

Dynamic Library Features 125

Performance Issues for Dynamic Library 126

Position-Independent Code and `-pic`. 126

Binding Options 127

A Simple Dynamic Library 128

Dynamic Library for Exporting Initialized Data. 131

7.5 Libraries Provided 135

8. Debugging 139
8.1 Global Program Checking (`-Xlist`) 139
Errors in General. 141
Details 141
Using Global Program Checking 142
Suboptions for Global Checking Across Routines 147
8.2 Special Compiler Options (`-C`, `-u`, `-U`, `-V`, `-xld`). 153
Subscript Bounds 153
Undeclared Variable Types 153
Case-sensitive Variable Recognition. 154
Version Checking 154
D Comment Line Debug Statements 155
8.3 The Debugger (`dbx`). 155
Sample Program for Debugging 156
Sample `dbx` Session 157
Segmentation Fault—Finding the Line Number. 159
Exception—Finding the Line Number 161
Bus Error—Finding the Line Number 162
Trace of Calls 163
Print Arrays. 164
Print Array Slices 165
Miscellaneous Tips 166
Main Features of the Debugger. 167
8.4 Debugging Parallelized Code 168

8.5 Compiler Messages in Listing (`error`) 168
Method 168
Usage of `error` Utility. 169
Options for `error` 169
Description 170
Sample Use of the `error` Utility 171
9. Floating Point 173
9.1 Summary 173
9.2 The General Problems 174
9.3 IEEE Solutions. 174
9.4 IEEE Exceptions 176
Detecting a Floating-point Exception. 176
Generating a Signal for a Floating-point Exception 176
Default Signal Handlers. 176
9.5 IEEE Routines 177
Flags and `ieee_flags()`. 178
Values and `ieee_values`. 183
Exception Handlers and `ieee_handler()`. 184
Retrospective. 194
Nonstandard Arithmetic 194
Messages about Floating-point Exceptions 195
9.6 Debugging IEEE Exceptions 195
9.7 Guidelines 196

9.8 Miscellaneous Examples 197
Kinds of Problems. 197
Simple Underflow. 198
Use Wrong Answer. 199
Excessive Underflow 199
10. Porting from Other FORTRANs. 203
10.1 General Hints 203
10.2 Time Functions 204
10.3 Formats 207
10.4 Carriage-Control. 207
10.5 File Equates 208
10.6 Data Representation. 208
10.7 Hollerith 209
10.8 Porting Steps. 211
Typical Case. 211
What To Do When Things Go Wrong. 213
11. Profiling 215
11.1 Summary 215
11.2 The `time` Command 216
iMPact FORTRAN MP Notes 217
11.3 The `gprof` Command 218
Compile and Link. 218
Execute 218
Run `gprof`. 218

11.4 The `tcov` Command 221
Compile and Link 221
Execute 222
Run `tcov` 222
List `p1.tcov` 222
11.5 I/O Profiling 223
11.6 Missing Profile Libraries 224
12. Performance 225
12.1 Introduction 225
12.2 Why Tune Code? 226
12.3 Algorithm Choice 226
12.4 Tuning Methodology 227
12.5 Loop Jamming 229
12.6 Benchmark Case History 230
Lessons 234
12.7 Optimizing 234
13. C–FORTRAN Interface 235
13.1 Sample Interface 235
13.2 How to Use this Chapter 236
13.3 Getting It Right 237
Function or Subroutine 238
Data Type Compatibility 239
Case Sensitivity 240
Underscore in Names of Routines 240

Passing Arguments by Reference or Value 241
Arguments and Order 241
Array Indexing and Order. 242
Libraries and Linking with the f77 Command 242
File Descriptors and stdio. 243
File Permissions 244
13.4 FORTRAN Calls C 245
Arguments Passed by Reference (f77 Calls C). 245
Arguments Passed by Value (f77 Calls C) 254
Function Return Values (f77 Calls C) 257
Labeled Common (f77 Calls C) 264
Sharing I/O (f77 Calls C) 265
Alternate Returns (f77 Calls C) - N/A 267
13.5 C Calls FORTRAN 268
Arguments Passed by Reference (C Calls f77). 268
Arguments Passed by Value (C Calls f77) - N/A 272
Function Return Values (C Calls f77) 273
Labeled Common (C Calls f77) 279
Sharing I/O (C Calls f77) 280
Alternate Returns (C Calls f77) 282
A. Runtime Error Messages 283
A.1 Operating System Error Messages 283
A.2 Signal Handler Error Messages 284
A.3 I/O Error Messages 284

B. XView Toolkit 289
B.1 XView Overview. 289
Tools 290
Objects 290
B.2 FORTRAN Interface. 290
Compiling 290
Initializing 291
Header Files. 292
Generic Procedures 293
Attribute Procedures 294
Attribute Lists 294
Handles 296
Data Types 296
Coding Fragment 296
B.3 C to FORTRAN 297
B.4 Sample Programs 299
B.5 References 301

C. iMPact: Multiple Processors 303
C.1 Requirements 303
C.2 Overview 304
Automatic Parallelization 304
Explicit Parallelizing 304
The `libthread` Primitives 304
Summary 305
Standards 306
C.3 Speed Gained or Lost 306
C.4 Number of Processors 307
C.5 Debugging Tips and Hints for Parallelized Code. 309
D. iMPact: Automatic Parallelization 313
D.1 What You Do 313
D.2 What the Compiler Does 314
Parallelize the Loop 314
Dependency Analysis. 315
Definitions: Array, Scalar, and Pure Scalar 315
D.3 Definition: Automatic Parallelizing 316
General Definition 316
Details 316
Exceptions for Automatic Parallelizing 317

D.4 Reduction for Automatic Parallelizing 320
What You Do 320
What the Compiler Does 320
Recognized Reductions 321
Roundoff and Overflow / Underflow for Reductions 322
E. iMPact: Explicit Parallelization 325
E.1 What You Do 325
E.2 What the Compiler Does 326
E.3 Parallel Pragma 327
E.4 `doall` Loops 327
Definition 327
Explicitly Parallelizing a `doall` Loop 328
`CALL` in a Loop 329
E.5 Exceptions for Explicit Parallelizing 329
Warning Messages by `-vpara` 330
I / O with Explicit Parallelization 332
E.6 Risk with Explicit: Nondeterministic Results 333
Testing is not Enough 333
How Indeterminacy Arises 334
E.7 Scope of Variables for Explicit Parallelizing 334
E.8 Signals 335
Index 337
Join the SunPro SIG Today 357

Figures

Figure 4-1 File System Hierarchy 78

Figure 4-2 Relative Path Name 80

Figure 4-3 Absolute Path Name 81

Tables

Table 1-1	New FORTRAN Features Since 3.0	5
Table 1-2	New FORTRAN Features Since 2.0 / 2.0.1	6
Table 3-1	File Name Suffixes FORTRAN Recognizes	24
Table 3-2	Options Frequently Used	25
Table 3-3	Actions / Options Available from `f77`	26
Table 3-4	Default Search Paths for `#include` Files	42
Table 7-1	*BaseDir* for Library Search Paths	120
Table 7-2	Some Major Libraries Provided	135
Table 9-1	`ieee_flags` Argument Meanings	179
Table 9-2	Functions for Using IEEE Values	183
Table 10-1	Time Functions Available to FORTRAN	204
Table 10-2	Summary: VMS FORTRAN System Routines	205
Table 10-3	Maximum Characters per Data Type	209
Table 13-1	Argument Sizes and Alignments—Pass by Reference	239
Table 13-2	Characteristics of Three I / O Systems	244
Table B-1	Matching C and FORTRAN Declarations	297

Table C-1 Parallelization Summary 305
Table D-1 Reductions Recognized by the Compiler. 321
Table E-1 Exceptions for Explicit Parallelizing. 330

Preface

This preface is organized into the following sections.

Purpose and Audience	*page xxi*
How This Book is Organized	*page xxii*
Related Documentation	*page xxiii*
Conventions in Text	*page xxvi*

Purpose and Audience

This guide shows *how to use* the Sun™ compiler FORTRAN 3.0.1.

It introduces the following aspects of this FORTRAN.

- Using the compiler command and options
- Global program checking across routines
- Using iMPact™ multiprocessor FORTRAN MP
- Debugging FORTRAN
- Using IEEE floating point with FORTRAN
- Making and using libraries
- Using some utilities and development tools
- Mixing C and FORTRAN
- Profiling and tuning FORTRAN

This guide is for scientists and engineers with the following background:

- Thorough knowledge and experience with FORTRAN programming
- General knowledge and understanding of some operating system
- Particular knowledge of SunOS™ or UNIX[1] commands `cd`, `pwd`, `ls`, `cat`.

It does not teach programming or the FORTRAN language. For details on a language feature or a library routine, see the *FORTRAN 3.0.1 Reference Manual.*

How This Book is Organized

This book is organized as follows.

Chapter 1, Introduction	*page 1*
Chapter 2, Getting Started	*page 17*
Chapter 3, Using the Compiler	*page 21*
Chapter 4, File System and FORTRAN I/O	*page 77*
Chapter 5, Disk and Tape Files	*page 85*
Chapter 6, Program Development	*page 99*
Chapter 7, Creating and Using Libraries	*page 113*
Chapter 8, Debugging	*page 139*
Chapter 9, Floating Point	*page 173*
Chapter 10, Porting from Other FORTRANs	*page 203*
Chapter 11, Profiling	*page 215*
Chapter 12, Performance	*page 225*
Chapter 13, C–FORTRAN Interface	*page 235*
Appendix A, Runtime Error Messages	*page 283*
Appendix B, XView Toolkit	*page 289*
Appendix C, iMPact: Multiple Processors	*page 303*
Appendix D, iMPact: Automatic Parallelization	*page 313*
Appendix E, iMPact: Explicit Parallelization	*page 325*
Index	*page 337*
Join the SunPro SIG Today	*page 357*

1. UNIX is a registered trademark of Novell, Inc., in the United States and other countries.

Related Documentation

The related kinds of documentation included with FORTRAN are as follows:

- Paper manuals (hard copy)
- On-line manuals in the AnswerBook™ viewing system
- On-line `man` pages
- On-line `README`s directory of information files and feedback form
- SunPro SIG (Sun Programmers Special Interest Group) publications and files
- IEEE and ISO POSIX.1 Standard (See *POSIX Library*, page 136.)

AnswerBook

The AnswerBook system displays and searches the on-line copies of the paper manuals. The system and manuals are included on the CD-ROM and can be installed to hard disc during installation. Installing and starting AnswerBook are described in the installation manual.

Related Manuals

The following documents are provided on-line or in hard copy, as indicated.

Title	Paper	AnswerBook
FORTRAN 3.0.1 User's Guide	X	X
FORTRAN 3.0.1 Reference Manual	X	X
Debugging a Program	X	X
Numerical Computation Guide	X	X
What Every Computer Scientist Should Know About Floating-Point Arithmetic		X
Installing SunSoft Developer Products on Solaris	X	X

The following documents are also relevant.

- *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, April 1978, American National Standards Institute, Inc.
- *Cray*[1] *Computer Systems* FORTRAN *(CFT) Reference Manual*, SR-0009

1. CRAY is a trademark of Cray Research Inc.

man *Pages*

Purpose

A man page is intended to answer the following questions:

- What does it do?
- How do I use it?

A man page serves as a:

- *Memory Jogger*— A man page *reminds* the user of details, such as arguments and syntax. It assumes you knew and forgot. It is not a tutorial.
- *Quick Reference*—A man page helps find something *fast*. It is brief, covering highlights. It is a *quick* reference, not a complete reference.

Usage

To display a man page, use the man command.

Example: Display the f77 man page.

```
demo$ man f77
```

Example: Display the man page for the man command.

```
demo$ man man
```

Operating System man *Pages and FORTRAN* man *Pages*

Some man pages have two versions—one for the operating system and one for FORTRAN. The default paths cause man to show the one for FORTRAN, but you can direct man to search in the operating system man pages directory first.

Example: One way to display the *operating system* man page for ctime.

```
demo$ man -M /usr/man ctime
```

The man command also uses the MANPATH environment variable, which can determine the set of man pages that are accessed. See man(1).

Related man *Pages*

The following man pages may be of interest to FORTRAN users.

man pages	Contents
f77(1)	Invoke the FORTRAN compiler
asa(1)	Print files having Fortran carriage-control
dbx(1)	Debug by a command-line-driven debugger
debugger(1)	Debug by a graphical-user-interface debugger
fsplit(1)	Print files having Fortran carriage-control
ieee_flags(3M)	Examine, set, or clear floating-point exception bits
ieee_handler(3M)	Exception handling
matherr(3M)	Error handling

READMEs

The READMEs directory has information files: bug descriptions, information discovered after the manuals were printed, feedback form, and so forth.

Location	Standard Installation	Nonstandard Installation to */my/dir/*
Solaris 1.x	/usr/lang/READMEs/	*/my/dir*/READMEs/
Solaris 2.x	/opt/SUNWspro/READMEs/	*/my/dir*/SUNWspro/READMEs/

Files	Contents
feedback	Sun programmers email template file: Send feedback comments to Sun
fortran	FORTRAN bugs, new features. behavior changes, documentation errata
ratfor.ps	Ratfor User's Guide. This is a PostScript[1] file. Print it with lp on any PostScript-compatible printer that has Palatino font. View it on-line with imagetool. (For *Solaris 1.x*, print with lpr, view with pageview.)

1. PostScript is a registered trademark of Adobe systems.

SIG

Sun Programmers Special Interest Group membership entitles you to other documentation and software. A membership form is included at the very end of this book. See "*Join the SunPro SIG Today*," on page 357.

Conventions in Text

We use the following conventions in this manual to display information.

- We show code listing examples in boxes.

```
      WRITE( *, * ) 'Hello world'
```

- The `plain typewriter font` shows prompts and coding.
- In dialogs, the **`boldface typewriter font`** shows text the user types in.

```
demo$ echo hello
hello
demo$ ■
```

- *Italics* indicate general arguments or parameters that you replace with the appropriate input. Italics also indicate emphasis.
- For Solaris 2.x, the default shell is `sh` and the default prompt is the dollar sign (`$`). Most systems have distinct host names, and you can read some of our examples more easily if we use a symbol longer than a dollar sign. Examples generally use "`demo$`" as the system prompt; where the `csh` shell is shown, we use "`demo%`" as the system prompt.
- The small clear triangle Δ shows a blank space where that is significant.

```
ΔΔ36.001
```

- We generally tag nonstandard features with a small black diamond (♦). Wherever we indicate that a feature is *nonstandard*, that means a program using it does not conform to the ANSI X3.9-1978 standard, as described in *American National Standard Programming Language* FORTRAN, ANSI X3.9-1978, April 1978, American National Standards Institute, Inc., abbreviated as the FORTRAN Standard.
- We usually show FORTRAN examples in tab format, not fixed column.
- We usually abbreviate "FORTRAN" as "`f77`".

Introduction 1

This chapter is organized into the following sections.

Operating Environments	*page 2*
Standards	*page 3*
Extensions	*page 3*
New Features and Behavior Changes	*page 5*
Compatibility	*page 13*
Text Editing	*page 14*
Program Development	*page 15*
Debugging	*page 15*
Licensing	*page 16*

The FORTRAN compiler comes with a programming environment, including certain operating system calls and support libraries. It integrates with powerful development tools, including SunSoft™ tools such as the Debugger, `make`, MakeTool, and SCCS. Some examples assume you installed the *Source Compatibility Package.*

iMPact™ and Workshop™

The compiler is available in various packages and configurations:

- Alone, or as part of a package, such as the FORTRAN Workshop
- With or without the iMPact MT/MP multiple processor package

1

1.1 Operating Environments

This guide describes FORTRAN 3.0.1 under Solaris® 1.x and 2.x for SPARC, and under Solaris 2.x for x86 operating environments. The previous major release was ported to Intel[1] 80386-compatible computers running Solaris 2.x for x86, and some features remain in this guide identified as being for x86 only.

- Most aspects of FORTRAN under 2.x, 1.x, and x86 are the same, including functionality, behavior, and features.
- Anything unique to one operating environment is tagged "(*2.x only*)" or sometimes "(*1.x*)" or "(*x86*)".
- The iMPact multiprocessor FORTRAN features are available only on SPARC, in Solaris 2.3, and later.

Definitions

- The Solaris 2.x operating environment includes (among other things):
  - Operating system: SunOS™ 5.*x*

    SunOS 5.*x* is based on the System V Release 4 (SVR4) UNIX operating system, and the ONC+™ family of published networking protocols and distributed services, including ToolTalk™.
  - Windows system: OpenWindows™ 3.x application development platform
- Solaris 1.x includes (among other things):
  - Operating system: SunOS 4.1.*x*

    SunOS 4.1.*x* is based on the UCB 4.3 BSD operating system.
  - Windows system: OpenWindows 3.x application development platform

Abbreviations

- Solaris 2.x is an abbreviation for "Solaris 2.2 and later."
- Solaris 1.x is an abbreviation for "Solaris 1.1.1 and later."
- SunOS 5.*x* is an abbreviation for "SunOS 5.2 and later."
- SunOS 4.1.*x* is an abbreviation for "SunOS 4.1.1, 4.1.2, 4.1.3, 4.1.3_U1."

1. Intel is a registered trademark of Intel Corporation.

1.2 Standards

This FORTRAN is an enhanced FORTRAN 77 development system. It:

- Conforms to the ANSI X3.9-1978 FORTRAN standard and the corresponding International Standards Organization number is ISO 1539-1980. NIST (formerly GSA and NBS) validates it at appropriate intervals.
- Conforms to the standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- Provides an IEEE standard 754-1985 floating-point package.
- Provides support on SPARC systems for optimization exploiting features of SPARC V8, including the SuperSPARC™ implementation[1]. These features are defined in the *SPARC Architecture Manual: Version 8*.

1.3 Extensions

This FORTRAN compiler provides the following features or extensions:

- Global program checking across routines for consistency of arguments, commons, parameters, etc.
- The iMPact multiprocessor FORTRAN package (*Solaris 2.x, SPARC only*)

  iMPact FORTRAN includes automatic and explicit loop parallelization, is integrated tightly with optimization, and requires a separate license.
- Many VAX[2] VMS FORTRAN 5.0 extensions, including:
  - `NAMELIST`
  - `DO WHILE`
  - Structures, records, unions, maps
  - Variable format expressions

  You can write FORTRAN programs with many VMS extensions so that these programs run with the same source code on both SPARC and VAX systems.
- Recursion
- Pointers
- Double-precision complex
- Quadruple-precision real (*SPARC only*)
- Quadruple-precision complex (*SPARC only*)

1. SuperSPARC is a trademarks of Texas Instruments, Inc.
2. VAX and VMS are a trademarks of Digital Equipment Corporation

Mixing Languages

On Solaris systems, routines written in C, C++, or Pascal can be combined with FORTRAN programs, since these languages have common calling conventions.

Optimization

This FORTRAN has global, peephole, and potential parallelization optimizations so you can create FORTRAN applications that are significantly faster and smaller. Benchmarks show that even without parallelization, optimized applications are as much as 460% faster, with an average of 10% reduction in code size when compared to unoptimized code. Refer to SIAM Newsletter, May 1989.

1.4 New Features and Behavior Changes

New Features in 3.0.1

For 3.0.1, the *major* new features are Solaris 1.x and global program checking. A summary of *all* new features is provided in the table below.

Table 1-1 **New FORTRAN Features Since 3.0**

New or Changed Feature			User Guide	Ref Manual
Added global program checking: `-Xlist` (arguments, commons, parameters, …)			*page 62* *page 139*	
Improved the `-Xlist` output format			*page 139, …*	
Added the following options:				
	`-nocx`	Smaller executable file—shrink by about 128K bytes (*1.x only*).	*page 45*	
	`-xlibmopt`	Use a library of selected math routines optimized for performance.	*page 60*	
	`-xnolibmopt`	Reset `-fast` so it does not use the library of selected math routines.	*page 60*	
	`-Zlp`	Prepare code for the loop profiler (*2.x, SPARC only*)	*page 62*	
	`-Ztha`	Prepare code for Thread Analyzer (*2.x, SPARC only*)	*page 63*	
Improved parallelization— do 25% more loops (private arrays, better fusion, …)			*page 316*	
Documented the `TMPDIR` environment variable (runtime scratch files put into the specified directory)				*Ch 4,* *`OPEN` statement*
Documented the `-unroll=`*n* option to do loop unrolling.			*page 57*	
Libraries: Made extensive bug fixes to the `XView` library bindings			*n/a*	
Added the `Xlib` library bindings to the installation CD			*page 135*	
Added the POSIX library to the installation CD (*2.x only*)			*page 135*	

1

New Features in 3.0

For 3.0, the *major* new feature is iMPact multiprocessor FORTRAN. A summary of *all* new features is provided in the table below.

Table 1-2 **New FORTRAN Features Since 2.0/2.0.1**

New or Changed Feature			User Guide	Ref Man
Added the optional iMPact multiprocessor FORTRAN package *(2.x, SPARC only*)			*page 313, …*	
Updated `etime`—for multiprocessors: it returns *wall clock* time, and `v(2)` is 0.0 *(2.x, SPARC only*)			*n/a*	*Ch. 7, dtime, etime*
Added checking for changing a constant at runtime. Trying to change a constant triggers a segmentation fault (`SIGSEGV`). In previous releases, these codes ran, but some had unpredictable answers and no warning.			*n/a*	*Ch 4, PARAMETER*
Improved array processing to allow better optimization			*n/a*	
Improved subscript checking of `-C` to check the range on each subscript individually (Previously, it checked the range of the array as a whole.)			*page 153*	
Improved the execution speed for optimized code			*n/a*	
Added the new multi-thread-safe FORTRAN library *(2.x, SPARC only*)			*page 135*	
Increased the default limit on number of continuation lines from 19 to 99				*Ch 1*
Improved compilation speed for:				
	Compiles with no optimization		*n/a*	
	Programs with a large number of symbols		*n/a*	
Eliminated the limit on symbol table size, and changed `-Nn` to do nothing			*page 47*	
Added the following options:				
	`-386`	*Generate code for 80386 (x86 only).*	*page 29*	
	`-486`	*Generate code for 80486 (x86 only).*	*page 29*	
	`-autopar`	*Parallelize automatically (Solaris 2.x, SPARC only).*	*page 31*	
	`-cg92`	*Generate code to run on SPARC V8 architecture (SPARC only).*	*page 32*	
	`-depend`	*Data dependencies, analyze loops (SPARC only).*	*page 33*	
	`-fsimple`	*Simple floating-point model.*	*page 38*	
	`-fstore`	*Force floating-point precision of expressions (x86 only).*	*page 38*	
	`-loopinfo`	*Loop info, show which loops are parallelized (Solaris 2.x, SPARC only).*	*page 42*	
	`-mt`	*Multithread safe libs, use for low level threads (Solaris 2.x, SPARC only).*	*page 44*	
	`-pentium`	*Generate code for pentium (x86 only).*	*page 50*	
	`-reduction`	*Reduction loops, analyze loops for reduction (Solaris 2.x, SPARC only).*	*page 53*	

Table 1-2 **New FORTRAN Features Since 2.0/2.0.1** *(Continued)*

New or Changed Feature			User Guide	Ref Man
	`-stackvar`	*Stack the local variables to allow better optimizing with parallelizing.*	*page 55*	
	`-vpara`	*Verbose parallelization, show warnings (Solaris 2.x, SPARC only).*	*page 57*	
	`-noautopar`, `-nodepend`, `-noexplicitpar`, `-noreduction` (*Solaris 2.x, SPARC only*)		*page 45, …*	
	`-nofstore`	*(x86 only)*	*page 46*	
	`-xa`, `-xcg`*year*, `-xlibmil`, `-xlicinfo`, `-xnolib`, `-xO`[*n*], `-xpg`, `-xsb`, `-xsbfast` (synonyms for compatibility with C)		*page 58, …*	
Added parallel pragma,`c$par doall`, explicit parallelization (*2.x, SPARC only*)			*page 327*	*Ch. 1, pragmas*
Deleted the option `-cg87` (*It was present for Solaris 1.x only*).			*page 8*	
Deleted descriptions for the following obsolete options. They do nothing, but they do not break `make` files in this release: `-66`, `-align _block_`, `-pipe`, `-r4`, `-w66`			*n/a*	

Other Software Changes that Affect FORTRAN

- Using the debugger requires new (SC3.0.1) debugger release
- Fix and continue: In debugger, fix a routine, compile only that one, link, run
- Watch points: In debugger, watch for any change to the value of a variable
- Some debugger commands changed. For a list, in `dbx`, type: `help changes`
- Optional multiple thread library, `libthread` (from SunSoft)
- Linker debug aids, see `ld`(1), try `-qoption ld -Dhelp` (*Solaris 2.3 only*)

Differences for FORTRAN in Solaris 2.x/1.x /x86

Most aspects of FORTRAN under 2.x, 1.x, and x86 are the same, including functionality, behavior, and features. But there are some differences.

The following is a summary of some of those differences:

- Multiprocessor FORTRAN is for Solaris 2.x for SPARC only
- The POSIX library is for Solaris 2.x only
- Some options are under Solaris 2.x only

  `-autopar`, `-dy`, `-dn`, `-explicitpar`, `-G`, `-h`, `-loopinfo`, `-noautopar`, `-nodepend`, `-noexplicitpar`, `-noreduction`, `-reduction`, `-R`, `-vpara`, `-xF`, `-xs`, `-Zlp`, `-Ztha`

- Some options are under Solaris 1.x only

  `-align`, `-bsdmalloc`, `-nocx`

- Some options are under Solaris x86 only

  `-386`, `-486`, `-fstore`, `-nofstore`, `-pentium`

- Procedures for building a dynamic shared library differ.
  See “Dynamic Libraries” on page 125.
- Calls, usage, and return codes of signal handlers differ.
  See “Exception Handlers and ieee_handler()” on page 184.
- Paths for shared libraries and installation are different. For installation:
  - Solaris 2.x: `/opt/SUNWspro/SC3.0.1`
  - Solaris 1.x: `/usr/lang/SC3.0.1`

  See also “Search Order for Library Search Paths” on page 120.

Behavior Changes

The behavior of some features has changed.

Sun 4/1xx and Sun 4/2xx Systems

Some older Sun workstations do not work with this compiler.

- Solaris 1.x

  Applications built with this compiler are incompatible with the Sun 4/1*xx* and Sun 4/2*xx* systems under Solaris 1.x.

- Solaris 2.x

  In principle, applications built with this compiler under Solaris 2.x should run on Sun 4/1*xx* and 4/2*xx* systems under Solaris 2.x, but too slowly for anyone to really want to do it.

Upgrading from 3.0

- The `-Xlist` option output includes error messages about any inconsistent arguments, commons, parameters, and so forth. Earlier versions of `-Xlist` output did not include these error messages.
- The `-Xlist` option output does not include and index. Earlier versions of `-Xlist` output did.

Upgrading from 2.0/2.0.1

If you are upgrading from FORTRAN 2.0/2.0.1, the following behavior changes may affect your programs. See also "Upgrading from 3.0," above.

- Possible slower loading: more global symbols than before

  To provide for the *fix and continue* feature, all local variables are available globally to the debugger in a way that requires that they be loaded at link time. This can increase load time.

- Trying to change a constant

  The 3.0/3.0.1 release improves runtime error checking by preventing the changing of a constant. Trying to change a constant triggers a SIGSEGV. In previous releases, such programs did run, but some produced unpredictable answers without warning.

  Example: Trying to change a constant.

```
      PARAMETER (arg=2.71828)
      CALL  sbrtn5 ( arg )
      ...
      END
      SUBROUTINE  sbrtn5 ( x )
      x = 3.14159
      RETURN
      END
```

  The above gets error message: POSSIBLE ATTEMPT TO MODIFY CONSTANT.

  The Workaround:

  - General: Do not change a constant. If you must change something, make it a variable, not a constant.
  - Specific: In the above example, change the PARAMETER statement to a DATA statement.

    That is, change: `PARAMETER (arg=2.71828)`
    To: `DATA arg/2.71828/`

- Number of processors for FORTRAN MP

  Number of processors requested by all programs (users) must not exceed the total number of processors available. This could seriously degrade performance.

  Example: If there are 4 processors on the system, and if each of 3 programs requests 2 processors, performance can be seriously degraded.

- Do not call `alarm()` from an MP program.

- Subscript checking at runtime with `-C`

  The subscript checking has been improved with this release. With `-C`, now each subscript of an array gets checked. Before, only the total offset was checked.

  Example: Program than ran with no error message, but now displays one.

```
      DIMENSION a(10,10)
      a(11,1) = 0
      END
```

- Debugging FORTRAN programs that use other languages

  If you debug FORTRAN programs that use other languages, you can use the new `dbx language` command.

  Sometimes `dbx` gets confused about which language it is debugging, and users get confused, too. The `language` command can fix both confusions.

  - If `dbx` gets confused about the language, you can tell `dbx` about the language. Type "`language fortran`" or "`language c`" or ...

    Example: Tell `dbx` which language.

```
(dbx) language fortran
(dbx)
```

  - If you get confused, ask `dbx` about the language. Type "`language`".

    Example: Ask `dbx` which language.

```
(dbx) language
(dbx) fortran
(dbx)
```

- Output from an exception handler is unpredictable

  If you make your own exception handler, avoid doing any FORTRAN output from it. If you must do some, then call abort right after the output. This reduces the risk of a system freeze. FORTRAN I/O from an exception handler amounts to recursive I/O. See next paragraph.

- Recursive I/O does not work reliably

  If you list a function in an I/O list, and if that function does I/O, then during runtime the execution freezes, or some other unpredictable problem arises. This risk exists independent of parallelization.

  Example: Recursive I/O that fails intermittently.

```
      PRINT *, x, f(x)
      END
      FUNCTION f(x)
      PRINT *, x
      RETURN
      END
```

  The Workaround—Avoid recursive I/O.

- IOINIT
  - The IOINIT routine ignores `CCTL`, `BZRO`, `APND`

    The Workaround—None
  - The IOINIT routine uses a different labeled common

    IOINIT communicates internal flags to the runtime I/O system. Previous releases put the internal flags into the labeled common

```
      COMMON /IOIFLG/ IEOF, ICTL, IBZR
```

    This is not a feature you would use intentionally, but if you had a labeled common named `IOIFLG`, it could have been a disaster.

    The current release uses the labeled common

```
      COMMON /__IOIFLG/ IEOF, ICTL, IBZR
```

The two leading underscores take this out of the user name space, so it is safer from accidental disasters. Names starting with underscores are reserved for the compiler.

- `dtime` & `etime` in iMPact FORTRAN MP

  `dtime` has always returned the CPU time. In MP, `dtime` returns the sum of all the CPU times, so `dtime` can return an unexpectedly large number. This breaks most megaflops calculations.

  The Workaround—Use `etime`, which was changed to return wall clock time in an MP program. Wall clock time does not break most megaflops calculations.

Upgrading from 1.4

If you are upgrading from FORTRAN 1.4, the following behavior changes may affect your programs, but see also “Upgrading from 2.0/2.0.1” on page 9” and “Upgrading from 3.0” on page 8.

- Debugging optimized code

  You can now compile with both `-g` and `-O` options. That is, you can now debug with optimized code.
  - If you have make files that rely on `-g` overriding `-O`, then you will have to revise those make files because `-g` does not override `-O`.
  - If you have make files that check for warning messages such as “`-g` overrides `-O`,” you will have to revise those make files.
  - The combination `-O4 -g` turns off the inlining that you usually get with `-O4`. A warning message is issued.

- Quadruple precision trigonometric functions

  The precision of PI matches what is used in the expression where PI occurs. It was restricted to 66 bits in the 1.4 release.

- Debugging block data subprograms

  There is a behavior change from FORTRAN 1.4 in debugging block data subprograms.

  The Symptom—If you are debugging a main program that uses a block data subprogram, then the debugger cannot find variables that are in the block data subprogram.

  The Fix—In the debugger, use the `func` command with the name of the block data subprogram.

Example: Program with block data.

```
      PROGRAM my_main
      COMMON /stuff/ x, y, z
      PRINT *, x
      END
      BLOCK DATA  init
      COMMON /stuff/ a/1.0/, b/2.0/, c/3.0/
      END
```

Debugging the above block data program.

In dbx: If you do:

```
(dbx) print a
```

then dbx cannot find a.

However, if you first do:

```
(dbx) func init
```

and next do:

```
(dbx) print a
```

then dbx finds a.

1.5 Compatibility

FORTRAN 2.0/2.0.1 (or earlier) *source* is compatible with FORTRAN 3.0/3.0.1, except for minor changes due to operating system changes and bug fixes.

FORTRAN 2.0/2.0.1 to 3.0/3.0.1

Executables (`a.out`), libraries (`.a`), and object files (`.o`) compiled and linked in FORTRAN 2.0/2.0.1 under Solaris 2.x are compatible with FORTRAN 3.0/3.0.1 under Solaris 2.x.

BCP: Applications from Solaris 1.x Run in 2.x

The Binary Compatibility Package must be installed for the executable to run.

- Executables compiled and linked in Solaris 1.x do run in Solaris 2.3 and later, but they do not run as fast as they do when compiled and linked under the appropriate Solaris release.
- Libraries (`.a`) and object files (`.o`) compiled and linked in FORTRAN 2.0.1 under Solaris 1.x are *not* compatible with FORTRAN 3.0.1 under Solaris 2.x.

In Solaris 2.x, Make Applications for 1.x

Under Solaris 2.x it is possible to make executables and libraries for Solaris 1.x, although it is not recommended. The Binary Compatibility Package must be installed for the compiler to do this correctly. To make it all work you must:

- Use the Solaris 1.x compiler in BCP mode.
- Use the Solaris 1.x linker (`ld`), with `-qpath` set to the path for the 1.x `ld`.
- Link with the Solaris 1.x libraries. If you get "`bad magic number`" messages, check the `-L` options and the `LD_LIBRARY_PATH` environment variable.

See "BCP Mode: How to Make 1.x Applications Under 2.x" on page 74.

1.6 Text Editing

Several text editors are available.

`vi` A traditional text editor for source programs is `vi` (vee-eye), the visual display editor, including `ed` and `ex`. For more information, read the `vi` manual.

`textedit` The `textedit` editor and other editors are available.

`emacs` For the `emacs` editor, and other editors not from Sun, read the Sun document *Catalyst™, a Catalog of Third-Party Software and Hardware*.

`xemacs` Xemacs is an Emacs editor that provides interfaces to the selection service and to the ToolTalk™ service.

The EOS package ("Era On Sparcworks") uses these two interfaces to provide simple yet useful editor integration with two SPARCworks tools: the SourceBrowser and the Debugger. Era is an earlier name of this editor.

It is available through the University of Illinois, by anonymous `ftp`, at `ftp.cs.uiuc.edu:/pub/era`

1.7 Program Development

asa This utility is a FORTRAN *output filter* for printing files that have FORTRAN carriage-control characters in column one. The UNIX implementation on this system does not use carriage-control since UNIX systems provide no explicit printer files. You use `asa` when you want to transform files formatted with FORTRAN carriage-control conventions into files formatted according to UNIX line-printer conventions. See `asa`(1).

fsplit This utility splits one FORTRAN file of several routines into several files, so that there is one routine per file.

gprof This utility profiles by procedure. For Solaris 2.x, when the operating system is installed, `gprof` is included if you do a *Developer Install*, rather than an *End User Install*; it is also included if you install the package `SUNWbtool`.

sbrowser The SourceBrowser is a source code and call graph browser that makes it easy to find occurrences of any symbol (in all source files, including header files). It is included with `dbx`.

tcov This utility profiles by statement.

1.8 Debugging

error A utility to insert compiler error messages at the offending source file line. For Solaris 2.x, when the operating system is installed, `error` is included if you do a *Developer Install*, rather than an *End User Install*; it is also included if you install the package `SUNWbtool`.

-Xlist An option to check across routines for consistency of arguments, commons, etc.

dbx An interactive symbolic debugger that understands this FORTRAN.

debugger A window, icon, mouse, and pointer interface to `dbx`; `debugger` is a graphical user interface.

1.9 Licensing

This compiler uses network licensing, as described in the manual *Installing SunSoft Developer Products on Solaris*.

If you invoke the compiler, and a license is available, the compiler starts. If no license is available, your request for a license is put on a queue, and your compile continues when a license becomes available. A single license can be used for any number of simultaneous compiles by a single user on a single machine.

Several licenses can be required and (depending on the package purchased) you need to do one or more of the following:

- For FORTRAN 3.0.1, purchase and install a Sun FORTRAN 3.0.1 license.
- For `dbx`, `debugger`, and so forth, purchase and install a Sun SPARCworks (or Proworks, ...) 3.0.1 license.
- For the iMPact multiprocessor FORTRAN features, purchase and install a separate Sun iMPact multiprocessor license.

Usually a Workshop includes all necessary licenses.

Getting Started 2

This chapter is organized into the following sections.

Summary	*page 17*
Compiling	*page 18*
Running	*page 18*
Renaming the Executables	*page 18*

This chapter gives a bare minimum on how to compile and run FORTRAN programs under Solaris. This chapter is for you if you know FORTRAN thoroughly and need to start writing programs in this FORTRAN immediately. Skip to Chapter 3, “Using the Compiler,” to learn more about it first.

2.1 Summary

Before you use this release of `f77`, it must be installed and licensed. Read ***Installing SunSoft Developer Products on Solaris***

Using this FORTRAN involves three steps:

- Write and save a FORTRAN program using a `.f`, `.for`, or `.F` file suffix.
- Compile and link this file using the `f77` command.
- Execute by typing the name of the executable file.

2

Example: This program displays a message on the screen.

```
demo$ cat greetings.f
      PROGRAM GREETINGS
      PRINT *, 'Real programmers hack FORTRAN!'
      END
demo$ ■
```

2.2 Compiling

Compile and link using the `f77` command as follows.

```
demo$ f77 -fast greetings.f
greetings.f:
MAIN greetings:
demo$ ■
```

In this example, `f77` compiles `greetings.f` and puts the executable code on the `a.out` file.

2.3 Running

Run the program by typing `a.out` on the command line.

```
demo$ a.out
 Real programmers hack FORTRAN!
demo$ ■
```

2.4 Renaming the Executables

It is awkward to have the result of every compilation on a file called `a.out`, since if such a file exists, it is overwritten. You can avoid this in two ways.

- After each compilation, use `mv` to change the name of `a.out`.

```
demo$ mv a.out greetings
demo$ ■
```

- On the command line, use `-o` to rename the output executable file.

```
demo$ f77 –o greetings -fast greetings.f
greetings.f:
MAIN greetings:
demo$ ■
```

The above command places the executable code on the `greetings` file.

Either way, run the program by typing the name of the executable file.

```
demo$ greetings
 Real programmers hack FORTRAN!
demo$ ■
```

At this point, read Chapter 3, “Using the Compiler,” for the compiler options, the extended features, and the summary of performance optimization. If you are not familiar with a UNIX file system, read Chapter 4, “File System and FORTRAN I/O,” or refer to any introductory UNIX book.

2

Using the Compiler 3

This chapter is organized into the following sections.

Compile Command	*page 22*
Language Preprocessor	*page 23*
Command-line File Names	*page 24*
Unrecognized Arguments	*page 24*
Compiler Options	*page 25*
Native Language Support	*page 63*
Compiler Directives	*page 67*
Miscellaneous Tips	*page 69*

The main use of this compiler is to compile source to make an executable:

> That is, translate source code files to an executable load module, an `a.out` file (by default, `f77` automatically invokes the linker)

Other uses are:

- Translate source code files to:
  - Relocatable binary (`.o`) files, for later linking by `f77` or `ld`
  - Dynamic shared library (`.so`) file
- Link `.o` files into an executable load module (`a.out`) file
- Show the commands built by the compiler, but do not execute.
- Check programs globally (across source files, functions, subroutines, ...)
- Simple check for ANSI standard conformance

3

3.1 Compile Command

Before you use this release of `f77`, it must be installed and licensed. Read ***Installing SunSoft Developer Products on Solaris***

The syntax of a *simple* compiler command is as follows.

`f77` [*`options`*] *`sfn`* **…**

where *sfn* is a FORTRAN source file name that ends in .f, .F, or .for, and *options* is one or more of the compiler options.

Example: A compile command with two files.

```
demo$ f77  growth.f  fft.f
```

Example: A compile command, same files, with some options.

```
demo$ f77  -g  -u  growth.f  fft.f
```

A *more general* form of the compiler command is as follows.

`f77` [*`options`*] *`fn`* **…** [**`-l`***`x`*]

- The *`fn`* is a file name (not necessarily of a FORTRAN source file). See Section 3.3, “Command-line File Names,” on page 24.
- The `-l`*x* is the option to link with library `lib`*x*`.a`.
- The `-l`*x* is *after* the list of file names. Always safer. Not always required.

The files and the results of compilations are linked (in the order given) to make an executable program, named (by default) `a.out` or with a name specified by the `-o` option.

Compile Link Sequence

With the above commands, if you successfully compile the files `growth.f` and `fft.f`, the object files `growth.o` and `fft.o` are generated, then an executable file is generated with the default name `a.out`.

The files `growth.o` and `fft.o` are not removed. If there is more than one object file (`.o` file), then the object files are not removed. This allows easier relinking if there is a linking error.

If the compile fails, you get an error message for each error, the `a.out` file is not generated, and the remaining `.o` files are not generated.

The general compiler driver `f77` does the following:

- Calls `f77pass1`, the FORTRAN front end
- Calls the code generator, and optionally the optimizer
- Calls `ld`, the linker, which generates the executable file

`growth.f` → `f77` → `f77pass1` → *optimizer/inliner* → *code generator* → `growth.o` → `ld` → `a.out`

The *optimizer / inliner* is optional.

Consistent Compile and Link

Be consistent with compiling and linking.

- If you compile and link in separate steps, and you *compile* any subprogram with any of these options:

  `–a`, `–autopar`, `–cg89`, `–cg92`, `-explicitpar`, `–fast`, `-misalign`, `–p`, `-parallel`, `–pg`, `-Zlp`, `-Ztha`

  then be sure to *link* with the same options.

  Example: Compile `sbr.f` with `–a` and `smain.f` without it. Just pass the `–a` to the linker with the final `f77` command.

```
demo$ f77 -c -a sbr.f
demo$ f77 -c smain.f
demo$ f77 -a sbr.o smain.o
```

- If you compile *one* subprogram with any of the options:

  `–dalign`, `–f`, `-fast`, `-r8`

  then compile *all* subprograms of that program with the same options.

3.2 Language Preprocessor

The `cpp` program is the C language preprocessor, which is invoked during the first pass of a FORTRAN compilation if the source file name has the `.F` extension. Its main uses for FORTRAN are for constant definitions and conditional compilation.

See `cpp`(1), or *–Dname*, page 34.

3

3.3 Command-line File Names

If a file name in the command line has any of the following suffixes, then the compiler recognizes it and takes the appropriate action. See Table 3-1.

`.f`, `.for`, `.F`, `.r`, `.s`, `.S`, `.il`, `.o`

If a file name has some other suffix, or no suffix, it is passed to the linker.

Table 3-1 File Name Suffixes FORTRAN Recognizes

Suffix	Language	Action
`.f`	FORTRAN	Compile FORTRAN source files, put object files in current directory; default name of object file is that of the source but with `.o` suffix.
`.for`	FORTRAN	Same as `.f`.
`.F`	FORTRAN	Apply the C preprocessor to the FORTRAN source file before FORTRAN compiles it.
`.r`	Ratfor	Process Ratfor source files before compiling.
`.s`	Assembler	Assemble source files with the assembler.
`.S`	Assembler	Apply the C preprocessor to the assembler source file before assembling it.
`.il`	Inline Expansion	Process inline expansion code template files. The compiler uses these to expand inline calls to selected routines. Since it's the compiler, not the linker, that does this, be sure to include these `.il` files in the compile command.
`.o`	Object Files	Pass object files through to the linker.

3.4 Unrecognized Arguments

Any arguments `f77` does not recognize are taken to be one of the following:

- Linker option arguments
- Names of `f77`-compatible object programs (maybe from a previous run)
- Libraries of `f77`-compatible routines

Note – Unrecognized *options* (with a "`-`") generate warnings.
Unrecognized *arguments* that are not options (no "`-`") generate no warnings; but if the linker does not recognize them, they generate error messages.

3.5 Compiler Options

This compiler has the power of many optional features, so there is a *long* list. To help with *the long list*, the options are listed from different perspectives:

● *Options/Actions Frequently Used*	*page 25*
● *Actions Summary—Actions/Options Sorted by Action (Actions and which option invokes them)*	*page 26*
(See also "compile action" in the Index.)	
● *Options/Actions Sorted by Option (Options and what they do)*	
All details: caveats, restrictions, interactions, and examples	*page 29*
Major details: Some caveats, restrictions, interactions, examples	`man f77`
Summary one-liner descriptions	`f77 -help`
Summary one-liner descriptions	*Quick Reference Card*
(See also the option name in the Index.)	

Actions/Options Frequently Used

A few options are needed by almost every programmer.

Table 3-2 Options Frequently Used

Action	Option	Details
Debug—global program checking across routines for consistency of arguments, commons, ...	`-Xlist`	*page 62*
Debug—produce additional symbol table information for the debugger.	`-g`	*page 39*
Performance—make executable run faster using a selection of options.	`-fast`	*page 36*
Performance—make executable run faster using the optimizer.	`-O[`*n*`]`	*page 49*
Library—bind (dynamically / statically) libraries listed *later* in command. `-Bdynamic`, `-Bstatic`	`-B`*binding*	*page 32*
Library—bind (dynamically / statically) libraries for executable. `-dy`, `-dn` (*Solaris 2.x only*).	`-d`*binding*	*page 34*
Compile only—suppress linking. Make a `.o` file for each source file.	`-c`	*page 32*
Name the final output file *outfil* instead of `a.out`.	`-o` *outfil*	*page 48*
Profile by procedure for `gprof`.	`-pg`	*page 51*

Be sure to check "*Details*" for caveats, restrictions, interactions, and examples.

3

Actions Summary (Actions/Options Sorted by Action)

Check "*Details*" for caveats, restrictions, interactions, and examples.

Table 3-3 Actions / Options Available from `f77`

	Action	Option	Details
Debug			
	Compile for use with the debugger.	`-g`	*page 39*
	Global checking—across routines (arguments, commons, parameters, ...).	`-Xlist`	*page 62*
	Subscript checking—runtime check for array subscripts out of range.	`-C`	*page 32*
	Undeclared variables—show a warning message.	`-u`	*page 57*
	Uppercase identifiers—leave in the original case.	`-U`	*page 57*
	Version ID—show version ID along with name of each compiler pass.	`-V`	*page 58*
	Debug statements—Use VMS debug print statements (`D` / `d` in column one).	`-xld`	*page 60*
	No autoload—disable autoload for `dbx` (*Solaris 2.x only*).	`-xs`	*page 61*
Floating Point			
	Use the best floating-point arithmetic for this machine.	`-native`	*page 45*
	Nonstandard arithmetic—allow nonstandard arithmetic.	`-fnonstd`	*page 37*
	`REAL` to `DOUBLE`—interpret `REAL` as `REAL*8`.	`-r8`	*page 53*
	Use simple model.	`-fsimple`	*page 38*
	Force precision of expressions (*x86 only*).	`-fstore`	*page 38*
	No forcing of expression precision (*x86 only*).	`-nofstore`	*page 46*
Library			
	Bind (dynamically / statically) all libraries listed later in the command.	`-B`*binding*	*page 32*
	Bind (dynamically / statically) for whole executable (*Solaris 2.x only*).	`-d`*binding*	*page 34*
	Build a dynamic shared library (*Solaris 2.x only*).	`-G`	*page 39*
	Directory—search this directory first.	`-L`*dir*	*page 43*
	Link with library `lib`*x*.	`-l`*x*	*page 43*
	Multithread safe libraries, low level threads (*Solaris 2.x, SPARC only*).	`-mt`	*page 44*
	Name a shared dynamic library (*Solaris 2.x only*).	`-h`*name*	*page 39*
	Paths—store into object file (*Solaris 2.x only*).	`-R` *list*	*page 54*
	No automatic libraries.	`-nolib`	*page 46*
	No inline templates.	`-nolibmil`	*page 46*
	No run path in executable (*Solaris 2.x only*).	`-norunpath`	*page 47*

Table 3-3 Actions / Options Available from `f77` *(Continued)*

	Action	Option	Details
Performance			
	Double load / store—allow `f77` to use double load / store (*SPARC only*).	`–dalign`	*page 33*
	Faster execution—make executable run faster using a selection of options.	`–fast`	*page 36*
	Generate code to run on generic SPARC architecture. (*SPARC only*).	`–cg89`	*page 32*
	Generate code to run on SPARC V8 architecture (*SPARC only*).	`-cg92`	*page 32*
	Inline templates—select best.	`-libmil`	*page 42*
	Inline the specified user routines to optimize for speed.	`-inline=`*rlst*	*page 40*
	Optimize for execution time.	`-O`[*n*]	*page 49*
	Unroll loops—direct the optimizer on unrolling loops *n* times.	`-unroll=`*n*	*page 57*
	Use selected math routines optimized for performance (*SPARC only*).	`-xlibmopt`	*page 60*
	Reset `-fast` so that it does not use `-xlibmopt` (*SPARC only*).	`-xnolibmopt`	*page 60*
	`malloc` speedup (*Solaris 1.x only*).	`-bsdmalloc`	*page 31*
	Generate code for 80386 (*x86 only*).	`-386`	*page 29*
	Generate code for 80486 (*x86 only*).	`-486`	*page 29*
	Generate code for pentium (*x86 only*).	`-pentium`	*page 50*
	Data dependencies, analyze loops (*SPARC only*).	`-depend`	*page 33*
Parallelize			
	Use `-autopar`, `-explicitpar`, `-depend` (*Solaris 2.x, SPARC only*).	`-parallel`	*page 50*
	Parallelize automatically (*Solaris 2.x, SPARC only*).	`-autopar`	*page 31*
	Parallelize explicitly (*Solaris 2.x, SPARC only*).	`-explicitpar`	*page 35*
	Parallelize reduction loops (*Solaris 2.x, SPARC only*).	`-reduction`	*page 53*
	Show which loops are parallelized, compile time (*Solaris 2.x, SPARC only*).	`-loopinfo`	*page 42*
	Profile all loops, after execution (*Solaris 2.x, SPARC only*).	`-Zlp`	*page 62*
	Stack local variables to allow better optimizing with parallelizing.	`-stackvar`	*page 55*
	Verbose—show warnings (*Solaris 2.x, SPARC only*).	`-vpara`	*page 57*
	No automatic parallelization (*Solaris 2.x, SPARC only*).	`-noautopar`	*page 45*
	No `-depend` (*SPARC only*).	`-nodepend`	*page 45*
	No explicit parallelization (*Solaris 2.x, SPARC only*).	`-noexplicitpar`	*page 45*
	No reduction (*Solaris 2.x, SPARC only*).	`-noreduction`	*page 47*
	Threads—analyze threads (*Solaris 2.x, SPARC only*).	`-Ztha`	*page 63*

Table 3-3 Actions / Options Available from `f77` *(Continued)*

	Action	Option	Details
Profile by			
	Basic block for `tcov`.	`-a`	*page 30*
	Procedure for `gprof`.	`-pg`	*page 51*
	Procedure for `prof`.	`-p`	*page 50*
Align			
	Align a common block on page boundaries. (*Solaris 1.x, SPARC only*).	`–align` *_block_*	*page 30*
	Align on 8-byte boundaries (*SPARC only*).	`-f`	*page 36*
	Allow for misaligned data (*SPARC only*).	`-misalign`	*page 44*
Licensing			
	License information—display license server user ids.	`-xlicinfo`	*page 60*
	No license queue.	`-noqueue`	*page 46*
Backward Compatibility			
	`DO` loops—use one trip `DO` loops.	`-onetrip`	*page 48*
	Output—use old list-directed output.	`-oldldo`	*page 48*
	Structures—use old structures.	`-oldstruct`	*page 48*
Miscellaneous			
	Assembly source—generate only assembly source code.	`-S`	*page 56*
	ANSI conformance—identify many non-ANSI extensions.	`–ansi`	*page 30*
	Command—show command line built by driver, but do not execute.	`–dryrun`	*page 34*
	Compile only.	`-c`	*page 32*
	Compiler component path—insert specified directory.	`-Qpath` *pathname*	*page 52*
	Include path—insert for preprocessor.	`-I`*loc*	*page 41*
	Integers, short—make short integers.	`-i2`	*page 40*
	Integers, standard—make standard integers.	`-i4`	*page 40*
	Line length—extend source line maximum length to 132 columns.	`–e`	*page 35*
	Options—display the list of options.	`–help`	*page 40*
	Output—rename file.	`-o` *outfil*	*page 48*
	Pass option list to program.	`-Qoption` *prog oplis*	*page 52*
	Position-independent code—produce.	`-pic`	*page 51*
	Position-independent code—produce with 32-bit addresses (*SPARC only*).	`-PIC`	*page 51*
	Preprocessor—use `cpp`.	`–F`	*page 38*

Table 3-3 Actions / Options Available from `f77` *(Continued)*

Action	Option	Details
Miscellaneous *(continued)*		
Preprocessor—define name for use by preprocessor.	`–D`*name*	*page 34*
Produce code of the specified type.	`-Qproduce` *typ*	*page 52*
Quiet compile, prompt only—reduce number of messages from compiler.	`-silent`	*page 55*
Reorder functions—enable reordering of functions in core (*Solaris 2.x only*).	`-xF`	*page 58*
Smaller executable file—shrink by about 128K bytes (*Solaris 1.x only*).	`-nocx`	*page 45*
SourceBrowser—compile fast for the SourceBrowser.	`-sbfast`	*page 55*
SourceBrowser—compile for the SourceBrowser.	`-sb`	*page 55*
Symbol table—strip executable of symbol table (prevents debugging).	`-s`	*page 55*
Table sizes—reset internal compiler tables.	`-N`[`cdlnqsx`]*k*	*page 47*
Temporary files—set directory to locate temporary files.	`-temp=`*dir*	*page 56*
Time for execution—display for each compilation pass.	`-time`	*page 57*
Verbose compile—print name of each compiler pass.	`-v`	*page 57*
VMS FORTRAN—include more VMS extensions.	`-xl`[`d`]	*page 59*
Warning suppression—do not show warnings.	`-w`	*page 58*

Options Details (Options/Actions Sorted by Option)

This section shows all `f77` options, including various restrictions, caveats, interactions, examples, and other details.

`-386` Generate code for 80386 (*x86 only*).

Generate code that exploits features available on Intel 80386 compatible processors. The default is `-386`.

`-486` Generate code for 80486 (*x86 only*).

Generate code that exploits features available on Intel 80486 compatible processors. The default is `-386`. Code compiled with `-486` does run on 80386 hardware, but it may run slightly slower.

-a Profile by basic block for `tcov`.

Insert code to count the times each basic block is run. This invokes a runtime recording mechanism that creates one `.d` file for every `.f` file (at normal termination). The `.d` file accumulates execution data for the corresponding source file. The `tcov`(1) utility can then be run on the source file to generate statistics about the program. `-pg` and `gprof` are complementary to `-a` and `tcov`.

If you compile and link in separate steps, and you compile with `-a`, then be sure to link with `-a`. You can mix `-a` with `-O`*n*; in some earlier versions `-a` overrode `-O`*n*. For another way, read *Performance Tuning an Application*.

-align _*block*_ Align a common block on page boundaries (*Solaris 1.x, SPARC only*).

Solaris 2.x Inert. The `-align` option is for compatibility with older versions, and is an inert option. It is recognized, so it does not break any old `make` files. *But it does not do anything.*

Solaris 1.x For the common block whose FORTRAN name is *block*, align it on a page boundary. Its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. The space is required between `-align` and *_block_*. This is a linker option.

Example: Do page-alignment for the common block named `buffo`.

```
demo% f77 -align _buffo_ growth.f
```

This option applies to *uninitialized* data only. If any variable of the common block is initialized in a `DATA` statement, then the block will not be aligned.

If you do *not* use the `-U` option, then use lowercase for the common block name. If you *do* use `-U`, then use the case of the common block name in your source code.

-ansi ANSI conformance check—identify many non-ANSI extensions.

-autopar Parallelize automatically (*Solaris 2.x, SPARC only*).

Find and parallelize appropriate loops for multiple processors. Do dependence analysis (analyze loops for inter-iteration data dependence) and do loop restructuring. If optimization is not at `-O3` or higher, raise it to `-O3`.

`-autopar` conflicts with `-g`.

Avoid `-autopar` if you do your own thread management. See note under *–mt*.

The `-autopar` option requires the iMPact FORTRAN multiprocessor enhancement package. To get faster code, this option requires a multiple processor system. On a single-processor system the generated code usually runs slower.

Example: Automatic parallelization (assumes you set number of processors).

```
demo$ f77 -autopar any.f
```

Before you use automatic parallelization, see Appendix C, "iMPact: Multiple Processors" and Appendix D, "iMPact: Automatic Parallelization."

Number of Processors—To request a number of processors, set the `PARALLEL` environment variable. The default is 1.

- Do not request more processors than are available.
- If `N` is the number of processors on the machine, then for a one-user, multiprocessor system, try `PARALLEL=N-1`.
- See Section C.4, "Number of Processors."

Linking—If you use `-autopar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe FORTRAN runtime library. If you use `-autopar` and compile and link in *separate* steps, then you must also link with `-autopar`.

-bsdmalloc `malloc` speedup (*Solaris 1.x only*).

Use the faster `malloc` from the library `libbsdmalloc.a`. This `malloc` is faster but less memory efficient. This option causes the following items to be passed to the linker *(SunOS 4.1.2 and 4.1.3 only)*.

```
-u _malloc /lib/libbsdmalloc.a
```

-B*binding* Bind (dynamically / statically) all libraries listed later in the command.

`-Bdynamic`: *Dynamic* binding, that is, shared libraries.
`-Bstatic`: *Static* binding, that is, nonshared libraries.

You can link some libraries statically and some dynamically by specifying `-Bstatic` and `-Bdynamic` any number of times on the command line. This is a loader option. The default is *dynamic*. No space is allowed between `-B` and *binding*. Library options must be *after* the `.f` and `.o` files if libraries are static.

-c Compile only.

Suppress linking. Make a `.o` file for each source file.

-C Subscript checking—runtime check for array subscripts out of range.

Check for subscripts outside the declared bounds. This helps catch some causes of the dreaded segmentation fault.

If `f77` detects such an out-of-range condition at compile time, it issues an error message and does not make an executable. If `f77` cannot determine such an out-of-range condition until runtime, it inserts range-checking code into the executable. Naturally this option can increase execution time, and the increase may vary from trivial to significant. Some developers debug with -C, then recompile without `-C` for the final production executable.

-cg89 Generate code to run on generic SPARC architecture (*SPARC only*).

Use a subset of the SPARC V8 instruction set. With `-cg89` and optimization, the instructions are scheduled for faster executables on a generic SPARC machine. Code compiled with `-cg89` does run on `-cg92` hardware.

-cg92 Generate code to run on SPARC V8 architecture (*SPARC* only).

Allow the use of the full SPARC V8 instruction set. In particular, allow the following instructions to be generated in-line:

`smul`, `smulcc`, `sdiv`, `sdivcc`, `umul`, `umulcc`, `udiv`, `udivcc`, `fsmuld`

With `-cg92` and optimization, the instructions are scheduled for faster executables on a SuperSPARC machine.

Code compiled with `-cg92` does run on older machines, but it may run slowly, as the new instructions added to SPARC V8 are emulated by traps to the operating system.

General Comments

- For SPARC systems, the default code generation option is `-cg89`.
- You can mix routines compiled `-cg89` with routines compiled `-cg92`; that is, you can have both kinds in one executable.
- Use `fpversion`(1) to tell which to use so the executables run faster: `-cg89` or `-cg92`. It may take about a minute to start the full report.
- For Solaris 1.x systems only, there was a `-cg87` option; it has been deleted. To determine its replacement, use `fpversion`, as described above.

`–dalign` Allow f77 to use double load / store (*SPARC only*).

Generate double load / store instructions wherever possible for faster execution. Using this option automatically triggers the `–f` option, which causes all double-precision and quadruple-precision data types (both real and complex) to be double aligned. With `-dalign`, you may not get ANSI standard FORTRAN alignment. It is a trade-off of portability for speed.

If you compile one subprogram with `-dalign`, compile all subprograms of the program with `–dalign`.

`-depend` Data dependencies, analyze loops (*SPARC only*).

Analyze loops for inter-iteration data dependencies and do loop restructuring. The `-depend` option is ignored unless you also use `-O3` or `-O4`. Dependence analysis is also included with `-autopar` or `-parallel`. The dependence analysis is done at compile time. On multiple-processor systems, execution is usually faster with `PARALLEL` ≥ 2.

Dependence analysis may help on single-processor systems. However, if you try `-depend` on single-processor systems, you must not use either `-autopar` or `-explicitpar`. If either of them is on, then the `-depend` optimization is done for multiple-processor systems.

The iMPact FORTRAN multiprocessor package is *not* required for `-depend`.

-dryrun Show commands built by driver, but do not execute.

-d*binding* Bind (dynamically / statically) libraries for whole executable (*Solaris 2.x only*).

`-dy`: The executable is *dynamically* bound, that is, shared libraries.
`-dn`: The executable is *statically* bound, that is, nonshared libraries.

The default is *y for yes (dynamic*). No space is allowed between `-d` and *binding*. This is a loader / linker option.

These apply to the whole executable file and can be used only once on the command line. Library options must be *after* the after `.f` and `.o` files if libraries are static.

-D*name* Define *name* for preprocessor.

`-D`*name*=*def*: Define *name* to be *def*
`-D`*name:* Define *name* to be `1`

For `.F` files only: Define *name* to be *def* using the C preprocessor, as if by `#define`. If no definition is given, the name is "1" is assigned the value 1.

Predefined Values

- The compiler version is predefined (in hex) in `__SUNPRO_F77`

  Example: For FORTRAN 3.0.1, `__SUNPRO_F77=0x301`

- The following values are predefined on appropriate systems:

  `__sparc`, `__unix`, `__sun`, `__i386`, `__SVR4`

  For instance, the value `__i386` is defined on systems compatible with the 80386 (including the 80486), and it is not defined on SPARC systems. You can use these values in such preprocessor conditionals as the following.

```
#ifdef __sparc
```

- From earlier releases, these values (with no underscores) are also predefined, but they may be deleted in a future release:

  `sparc`, `unix`, `sun`, `i386`

-e Extend the source line maximum length to 132 columns.

Accept lines up to 132 characters long.

-explicitpar Parallelize explicitly (*Solaris 2.x, SPARC only*).

You do the dependency analysis: analyze and specify loops for inter-iteration data dependencies. The software parallelizes the specified loops. If optimization is not at `-O3` or higher, then it is raised to `-O3`. Avoid `-explicitpar` if you do your own thread management. See *–mt*.

`-explicitpar` conflicts with `-g`.

The `-explicitpar` option requires the iMPact FORTRAN multiprocessor enhancement package. To get faster code, this option requires a multiprocessor system. On a single-processor system the generated code usually runs slower. Before you parallelize *explicitly*, see Appendix C, "iMPact: Multiple Processors," Appendix D, "iMPact: Automatic Parallelization," and Appendix E, "iMPact: Explicit Parallelization."

Summary: To parallelize explicitly, do the following.

- Analyze the loops to find those that are safe to parallelize.
- Insert "`c$par doall`" to parallelize a loop.
- Use the `-explicitpar` option.

Example: Insert a parallel pragma immediately before the loop.

```
       ...
c$par doall
      do i = 1, n
          a(i) = b(i) * c(i)
      end do
       ...
```

Example: Compile to explicitly parallelize.

```
demo$ f77 -explicitpar any.f
```

Incorrect Results. Do not apply an explicit parallel pragma to a reduction loop. The explicit parallelization is done, but the reduction aspect of the loop is not done, and the results can be incorrect. The results of the calculations can even be *indeterminate*: you can get incorrect results, possibly different ones with each run, and with no warnings.

Linking—If you use `-explicitpar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe FORTRAN runtime library. If you use `-explicitpar` and compile and link in *separate* steps, then you must also *link* with `-explicitpar`.

`-f` Align on 8-byte boundaries (*SPARC only*).

Align all `COMMON` blocks and all double-precision and quadruple-precision local data on 8-byte boundaries. This applies to both real and complex data. Resulting code may not be standard and may not be portable. If you compile with `-f` for *any* subprogram of a program, then compile *all* subprograms of that program with `-f`.

`-fast` Faster execution—make executable run faster using a selection of options.

Select options that optimize for speed of execution without excessive compilation time. This provides close to the maximum performance for most realistic applications. `-fast` selects the following options.

- The `-native` best floating-point option

  If the program is intended to run on a different target than the compilation machine, follow the `-fast` with a code-generator option. For *SPARC*, an example is: `-fast -cg89`

- The `–O3` optimization level option

  For subprograms that benefit from `–O4`, follow `–fast` with `-O4`: "`-fast -O4`" Note that the "`-fast -O4`" pair does not do the same thing as the "`-O4 -fast`" pair. The last specification of each pair takes precedence.

  Example: Overriding part of `-fast` (note warning message).

```
demo$ f77 -silent  -fast -O4  forfast.f
f77: Warning: -O4 overwrites previously set optimization level of -O3
demo$ ■
```

- The `-libmil` option for system-supplied inline expansion templates

  For C functions that depend on exception handling specified by SVID (as do some `libm` programs), follow `-fast` by `-nolibmil`: "`-fast –nolibmil`". With `-libmil`, exceptions cannot be detected with `errno` or `matherr`(3m).

- The –fnonstd floating-point initialization option

  –fnonstd is unsuitable for subprograms that depend on IEEE standard exception handling. You can get different numerical results or premature program termination by unexpected SIGFPE signals.

- The -fsimple option for a simple floating-point model

  –simple is unsuitable if strict IEEE 754 standards compliance is required.

- The –dalign option to generate double loads and stores (*SPARC only*)

  This may not get ANSI standard FORTRAN alignment.

- The -xlibmopt option (*SPARC only*).

- For x86 only, the -nofstore option, so it does not force floating-point expressions to the precision of the destination variable

If you compile and link in separate steps, and you compile with -fast, then be sure to link with -fast.

-flags Synonym for -help.

–fnonstd Allow nonstandard arithmetic.

Allow nonstandard initialization of floating-point arithmetic hardware:

- Abort on exceptions
- Flush denormalized numbers to zero if it will improve speed

Where *x* does not cause total underflow, *x* is a *denormalized number* if and only if |x| is in one of the ranges indicated:

Data Type	Range
REAL	0.0 < |x| < 1.17549435e-38
DOUBLE PRECISION	0.0 < |x| < 2.22507385072014e-308

See the *Numerical Computations Guide* for details on denormalized numbers.

The standard initialization of floating-point is the default:

- IEEE 754 floating-point arithmetic is nonstop
- Underflows are gradual

Specifying `–fnonstd` during the link step is approximately equivalent to the following two calls at the beginning of a FORTRAN main program.

```
      i=ieee_handler("set", "common", SIGFPE_ABORT)
      call nonstandard_arithmetic()
```

The `nonstandard_arithmetic()` routine is equivalent to the obsolete `abrupt_underflow()` routine.

On some implementations, the `nonstandard_arithmetic()` call causes all underflows to produce zero rather than a possibly subnormal number, as the IEEE standard requires. This may be faster. See `ieee_functions`(3m).

The `–fnonstd` option allows hardware traps to be enabled for floating-point overflow, division by zero, and invalid operation exceptions. These are converted into SIGFPE signals, and if the program has no SIGFPE handler, it terminates with a dump of memory to a `core` file. See `ieee_handler`(3m).

`-fsimple` Simple floating-point model.

Allow optimization using mathematically equivalent expressions. The optimizer is allowed to act as if a simple floating-point model holds during compilation and runtime. It is allowed to optimize without regard to roundoff or numerical exceptions.

With `-fsimple`, the optimizer can assume the following:

- The IEEE 754 default rounding and trapping modes hold.
- No exceptions arise other than inexact.
- The program does not test for inexact exceptions.
- There are no infinities or NaNs.
- The program does not depend on the sign of zero.

`-fstore` Force floating-point precision of expressions (*x86 only*).

Use the precision of destination variable. This applies to assignment statements only. Default (unless `-fast` is on) `-fstore`.

`–F` C preprocessor, `cpp`.

Apply the CPP preprocessor to relevant files and put the result in the file with the suffix changed to `.f`, but do not compile.

-g Debug—compile for use with the debugger.

Produce a symbol table of information for the debuggers. If you plan to debug, you get more debugging power if you compile with `-g` before using the debuggers. The `–g` option suppresses the automatic inlining you usually get with `–O4,` but does not suppress `–O`*n*.

`-autopar`, `-explicitpar`, and `-parallel` conflict with `-g`.

`–O`*n* limits `–g` in such ways as the following:

- Local variables cannot be printed (optimizer can put them on the stack)
- Cannot step through a routine line by line (optimizer can change the order)

You can get around some `-O`*n* limits on `-g` in either of two ways:

- Recompile all routines with `-O1` or no `-O`*n* at all
- Recompile only the routine you need to debug using *fix and continue*

If upgrading from prior to 2.0, note the following side effects of `-g` not suppressing `–O`*n*:

- Old `Make` files that rely on `-g` overriding `-O`, must be changed.
- Old `Make` files that check for warning "`-g` overrides `-O`," must be changed.

-G Library—build a dynamic shared library (*Solaris 2.x only*).

Tell the linker to build a *shared dynamic* library. Without `-G`, the linker builds an executable file. With `-G`, it builds a dynamic library.

-h*name* Library—name a shared dynamic library (*Solaris 2.x only*).

The `-h`*name* option assigns a name to a shared dynamic library. This allows versions of a shared dynamic library. A space between `-h` and *name* is optional.In general, this *name* must be the same as what follows the `-o`.

This is a loader option. The compile-time loader assigns the specified name to the shared dynamic library being created, and it records the name in the library file as the *internal* name of the library.

If there is no `-h`*name* option, then no internal name is recorded in the library file. Every executable file has a list of needed shared library files. When the runtime linker links the library into an executable file, the linker copies the internal name from the library into that list of needed shared library files.

If there is no internal name of a shared library, then the linker copies the path of the shared library file instead.

Example: One way to use the `-h` option.

1. First make and use one version of a shared library.

```
demo$ ld -G -o libxyz.1 -h libxyz.1 ...  {create shared library}
demo$ ln libxyz.1 libxyz.so              {make link libxyz.so to libxyz.1}
demo$ f77 ...-o verA -lzyz  ...          {executable verA needs  libxyz.1}
```

2. Then make and use a second version of the library.

```
demo$ ld -G -o libxyz.2 -h libxyz.2 ...  {create shared library}
demo$ rm libxyz.so                       {remove old link}
demo$ ln libxyz.2 libxyz.so              {make link libxyz.so to libxyz.2}
demo$ f77 ...-o verB -lxyz  ...          {executable verB needs  libxyz.2}
```

`-help` Options—display the list of options.

Display an equivalent of this list of options and show how to send feedback comments to Sun.

`-i2` Integers, short—make short integers.

Make 2 the default size in bytes for integer and logical constants and variables. But for `INTEGER*`*n* `Y`, the `Y` uses *n* bytes, regardless of the `-i2`. This option may increase runtime. If you need short integers, it is generally better to use `INTEGER*2` for specific (large) arrays.

`-i4` Integers, standard—make standard integers.

Make 4 the default size in bytes for integer and logical constants and variables. In "`INTEGER Y`" the `Y` use 4 bytes, but in "`INTEGER*`*n* `Y`", the `Y` uses *n* bytes, regardless of `-i4`.

`-inline=`*rlst* Inline the specified user routines to optimize for speed.

Inline the user-written routines named in the list *rlst*. The list is a comma-separated list of functions and subroutines. Only routines in the file being compiled are considered. The optimizer decides which of these routines are appropriate for inlining.

Example: Inline the routines *sub1*, *sub6*, *sub9*.

```
demo$ f77 -O3 -inline=sub1,sub6,sub9 *.f
```

If compiling with `-O3`, the `-inline` option can increase optimization by inlining some routines. The `-O3` option inlines none by itself.

If compiling with `-O4`, the `-inline` option can decrease optimization by restricting inlining to only those routines in the list. With `-O4`, the optimizer normally tries to inline all appropriate user-written subroutines and functions.

A routine is not inlined if any of the following apply (no warnings issued):

- Optimization is less than `-O3`
- The routine cannot be found
- Inlining the routine does not look profitable or safe
- The source for the routine is not in the file being compiled

`-I`*loc* Include path—insert the include path for the preprocessor.

Insert the path *loc* at the start of the list of directories in which to search for preprocessor `#include` files. No space is allowed between `-I` and *loc*. Invalid directories are just ignored with no warning message.

The `-I`*loc* option does *not* affect the FORTRAN `INCLUDE` statement.

Example: `f77 -I/usr/applib growth.F`

Above, `f77` searches for `#include` files in the `/usr/applib` directory.

Use `-I`*loc* again to insert more paths. Example: `f77 -Ipath1 -Ipath2 any.F`

This path and directory list applies to the following:

- Those "`#include`" files with *relative* path names, not absolute path names
- The `#include` directive
- The `.F` files, and only `.F` files

Search Order: The search for `#include` files is in these places, in this order:

- The directory containing the source file
- Directories named in `-I`*loc* options
- Directories in the *default list*

The *default list* for –I*loc* depends on the directory for `f77` installation. This list is usually set to the following list of paths.

Table 3-4 Default Search Paths for #`include` Files

	Standard Install	**Nonstandard Install to** */my/dir/*
Solaris 1.x	`/usr/lang/SC3.0.1/include/f77` `/usr/include`	*/my/dir*`/SC3.0.1/include/f77` `/usr/include`
Solaris 2.x	`/opt/SUNWspro/SC3.0.1/include/f77/` `/usr/include/`	*/my/dir*`/SUNWspro/SC3.0.1/include/f77/` `/usr/include/`

`-Kpic` Synonym for `-pic`.

`-KPIC` Synonym for `-PIC`.

`-libmil` Inline templates, select best.

Select the best inline templates for the floating-point option and operating system release available on this system.

`-loopinfo` Loop information—show which loops are parallelized (*Solaris 2.x, SPARC only*).

Show which loops are parallelized and which are not. Use with the `-parallel`, `-autopar`, or `-explicitpar` options.

This option requires the iMPact FORTRAN multiprocessor enhancement package.

Pass `f77` standard error into the `error` utility to get an annotated source listing (each loop tagged as parallelized or not); otherwise loops are identified only by line number.

Example: `-loopinfo`, in `sh`, pass `f77` standard error to the `error` utility.

```
demo$ f77 -autopar -loopinfo any.f 2>&1 | error options
```

For details on `error`, see Section 8.5, “Compiler Messages in Listing (error).”

-l*x* Link with library `lib`*x*.

Pass "`-l`*x*" to the linker. `ld` links with object library `lib`*x*. If shared library `lib`*x*`.so` is available, `ld` uses it, otherwise, `ld` uses archive library `lib`*x*`.a`. If it uses a shared library, the name is built in to `a.out`. No space is allowed between `-l` and *x* character strings.

Example: Link with the library `libV77`.

```
demo$ f77 any.f -lV77
```

Use `-l`*x* again to link with more libraries.

Example: Link with the libraries `liby` and `libz`.

```
demo$ f77 any.f -ly -lz
```

See also Section 7.2, "Library Search Paths and Order," on page 117.

-L*dir* Library directory—search this directory first.

Add *dir* at the *start* of the list of object-library search directories. While building the executable file, `ld`(1) searches *dir* for archive libraries (`.a` files) and shared libraries (`.so` files). A space between `-L` and *dir* is optional. The directory *dir* is not built in to the `a.out` file. See also *–lx*. `ld` searches *dir* before the default directories. See "Search Order for Library Search Paths" on page 120. For the relative order between `LD_LIBRARY_PATH` and `-L`*dir*, see `ld`(1).

Example: Use `-L`*dir* to specify a library search directory.

```
demo$ f77 -Ldir1 any.f
```

Example: Use `-L`*dir* again to add more directories.

```
demo$ f77 -Ldir1  -Ldir2  any.f
```

Restrictions

Solaris 2.x and 1.x

- No `-L/usr/lib`: In Solaris 2.x and 1.x, do *not* use `–L`*dir* to specify `/usr/lib`. It is searched by default. Including it here may prevent using the unbundled `libm`.

Solaris 2.x

- No `-L/usr/ccs/lib`: In Solaris 2.x, do not use `–L`*dir* to specify `/usr/ccs/lib`. It is searched by default. Including it here may prevent using the unbundled `libm`.

`-misalign` Misaligned data—allow for misaligned data (*SPARC only*).

The `-misalign` option allows for misaligned data in memory. Use this option *only* if you get a warning that `COMMON` or `EQUIVALENCE` statements cause data to be misaligned. This option generates much slower code for references to dummy arguments. If you can, recode the indicated section instead of recompiling with this option.

Example: `-misalign`. The following program has misaligned variables.

```
      INTEGER*2   I(4)
      REAL        R1, R2
      EQUIVALENCE (R1, I(1)), (R2, I(2))
      END
```

The above program causes the following error message.

```
"misalign.f", line 4:  Error: bad alignment for "r2" forced by equivalence
```

If you compile and link in separate steps, and you compile with the `-misalign` option, then be sure to link with the `-misalign` option.

`–mt` Multithread safe libraries—use for low level threads (*Solaris 2.x, SPARC only*).

Use multithread safe libraries. If you do your own low level thread management this helps prevent conflicts between threads. For FORTRAN, the multithread safe library is `libF77_mt`.

Use `-mt` if you mix C and FORTRAN and you do your own thread management of multithread C coding using the `libthread` primitives. Before you use your own multi-threaded coding, read the "*Guide to Multi-Thread Programming*."

The `–mt` option does not require the iMPact FORTRAN multiprocessor enhancement package, but to compile and run it does require Solaris 2.2 or later. The equivalent of `-mt` is included automatically with `-autopar`, `-explicitpar`, or `-parallel`.

On a single-processor system the generated code can run slower with the `–mt` option, but not usually by a significant amount.

Restrictions

- With `-mt`, if a function does I/O, do not name that function in an I/O list. Such I/O is called *recursive* I/O, and it causes the program to hang (deadlock). Recursive I/O is unreliable anyway, but is more apt to hang with `-mt`.
- In general, do *not* combine your own multi-threaded coding with `-autopar`, `-explicitpar`, or `-parallel`. Either do it all yourself or let the compiler do it. You may get conflicts and unexpected results if you and the compiler are both trying to manage threads with the same primitives.

`–native` Native floating point—use what is best for this machine.

Direct the compiler to decide which floating-point options are available on the machine the compiler is running on, and generate code for the best one. If you compile and link in separate steps, and you compile with the `-native` option, then be sure to link with `-native`.

`-noautopar` No automatic parallelization (*Solaris 2.x, SPARC only*).

Do not parallelize loops automatically. This option requires the FORTRAN multiprocessor enhancement package.

`-nocx` Make the executable file smaller by about 128K bytes (*Solaris 1.x only*).

The smaller files are from not linking with `-lcx`. The runtime performance and accuracy of binary-decimal base-conversion will be somewhat compromised.

`-nodepend` No `-depend` (*SPARC only).*

Cancel any `-depend` from earlier on the command line. This option does not require the iMPact FORTRAN multiprocessor package.

`-noexplicitpar` No explicit parallelization (*Solaris 2.x, SPARC* only).

Do not parallelize loops explicitly. This option requires the iMPact FORTRAN multiprocessor package.

-nofstore No forcing of expression precision *(x86 only)*.

Do not force expression precision to precision of destination variable (*x86 only*). This is for assignment statements only. Default if `-fast` is on: `-nofstore`.

-nolib No automatic libraries.

Do *not* automatically link with *any* system or language library; that is do *not* pass any `-l`*x* options on to `ld`. The default is to link such libraries into the executables automatically, without the user specifying them on the command line.

The `-nolib` option makes it easier to link one of these libraries statically. The system and language libraries are required for final execution. It is the users responsibility to link them in manually. This provides complete control (and with control comes responsibility) for the user.

For example, a program linked dynamically with `libF77` fails on a machine that has no `libF77`. When you ship your program to your customer, you can ship `libF77` or you can link it into your program statically.

Example: Link `libF77` statically and link `libc` dynamically.

```
demo$ f77 -nolib any.f -Bstatic -lF77 -Bdynamic -lm -lc
```

There is no dynamic `libm`; it is always linked statically.

Order for `-l`*x* options is important. Use the order shown in the example.

–nolibmil No inline templates.

Reset `–fast` so that it does *not* include inline templates. Use this *after* the `-fast` option.

Example:

```
demo$ f77 –fast –nolibmil …
```

–noqueue No license queue.

If you use this option, and no license is available, the compiler returns without queueing your request and without doing your compile. A nonzero status is returned for testing in `make` files.

-noreduction No reduction (*Solaris 2.x, SPARC only*).

Do no reduction. This option requires the iMPact FORTRAN multiprocessor enhancement package.

-norunpath No run path in executable (*Solaris 2.x only).*

If an executable file uses shared libraries, then the compiler normally builds in a path that tells the runtime linker where to find those shared libraries. The path depends on the directory where you installed the compiler. The `-norunpath` option prevents that path from being built in to the executable.

This option is helpful if you have installed in some nonstandard location, and you ship an executable to your customers, but you do not want to make the customers deal with that nonstandard location. Compare with `-R`*list*.

–N[cdlnqsx]*k* Table sizes—reset internal compiler tables.

Make static tables in the compiler bigger. The compiler complains if tables overflow and suggests you apply one or more of these flags. No spaces are allowed within this option string. These flags have the following meanings:

-Nc*k*

Control statements. Maximum depth of nesting for control statements such as `DO` loops. The default is 25. Example: `f77 -Nc50 any.f`

-Nd*k*

Data structures. Maximum depth of nesting for data structures and unions. The default is 20. Example: `f77 -Nd30 any.f`

-Nl*k*

Continuation lines. Maximum number of continuation lines for a continued statement. The default is 99 (1 initial and 99 continuation). Any number greater than 19 is nonstandard. ♦ Example: `f77 -Nl200 any.f`

-Nn*k*

Identifiers. This option has no effect. The number of identifiers is now unlimited. This option is still recognized so it does not break make files, but it may be deleted in a future release.

-Nq*k*

Equivalence. Maximum number of equivalenced variables. The default is 500. Example: `f77 -Nq600 any.f`

-Ns*k*

Statement numbers. Maximum number of statement labels. The default is 2000. Example: `f77 -Ns3000 any.f`

-Nx*k*

External names. Maximum number of external names (common block names, subroutine and function names). The default is 1000. `Example: f77 -Nx2000 any.f`

–o *outfil* Output file—rename the output file.

Name the final output file *outfil* instead of `a.out`. There must be a blank between `-O` and *outfil*.

-oldldo Output—use old list-directed output.

Omit the blank that starts each record for list-directed output. This is a change from releases 1.4 and earlier. The default behavior is to provide that blank, since the FORTRAN Standard requires it. Note also the `FORM='PRINT'` option of `OPEN`. You can compile some source files with `-oldldo` and some without, in the same program.

-oldstruct Structures—use old structures.

Align structures as in releases 1.4 and earlier. Use this to read unformatted files created with these older versions of `f77`, which used an implementation slightly different from either the default way or the VMS (`-xl`) way. If you need to share structures with C, use the default; do not use `-xl` and do not use `-oldstruct`. If you use both `-oldstruct` and `-xl`, you get `-oldstruct`.

–onetrip `DO` loops—use one trip `DO` loops.

Compile `DO` loops so that they are performed at least once if reached. `DO` loops in this FORTRAN are not performed at all if the upper limit is smaller than the lower limit, unlike *some* implementations of FORTRAN 66 `DO` loops.

-O[*n*] Optimize for execution time.

The *n* can be 1, 2, 3, or 4. No space is allowed between `-O` and *n*. If `-O`[*n*] is not specified, the compiler still performs a default level of optimization; that is, it executes a single iteration of local common subexpression elimination and live/dead analysis.

`-g` does not suppress `-O`*n*, but `-O`*n* limits `-g` in certain ways; see *–g*, on page 39. For `makefile` changes regarding `-o` with `-g`, see *–g*, on page 39.

-O If you do not specify an *n*, `f77` uses whatever *n* is most likely to yield the fastest performance for most reasonable applications. For the current release of FORTRAN, this is 3.

-O1 Do only the minimum amount of optimization (peephole). This is postpass assembly-level optimization. Do not use `-O1` unless `-O2` and `-O3` result in excessive compilation time, or running out of swap space.

-O2 Do basic local and global optimization. This level usually gives minimum code size. The details are: induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, and complex expression expansion.

Do not use `-O2` unless `-O3` results in excessive compilation time, running out of swap space, or excessively large code size.

-O3 Besides what `-O2` does, this also optimizes references and definitions for external variables. Usually `-O3` makes larger code.

-O4 Besides what `-O3` does, this also does automatic inlining of routines contained in the same file. This usually improves execution speed, but sometimes makes it worse. Usually `-O4` makes larger code.

For most programs: `-O4` is faster than `-O3` is faster than `-O2` is faster than `-O1`. But in a few cases `-O2` might beat the others, or `-O3` might beat `-O4`. You can try compiling with each level to find if you have one of these rare cases.

The `-g` option suppresses the `-O4` automatic inlining described above.

The -O3 and -O4 options reduce the utility of debugging in that you cannot print variables, but you can use the dbx where command to get a symbolic traceback, without the penalty of turning off optimization.

SPARC only: If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent routines at the original level specified in the command-line option.

-p Profile by procedure for prof.

Prepare object files for profiling, see prof (1). If you compile and link in separate steps, and if you compile with the -p option, then be sure to link with the -p option. -p with prof are provided mostly for compatibility with older systems. -pg with gprof do more.

-parallel Parallelize—use -autopar, -explicitpar, -depend (*Solaris 2.x, SPARC only*).

Parallelize loops both automatically by the compiler and explicitly specified by the programmer. With explicit parallelization of loops, there is a risk of producing incorrect results. If optimization is not at -O3 or higher, then it is raised to -O3.

-parallel conflicts with -g.

Avoid -parallel if you do your own thread management. See *–mt*.

The -parallel option requires the iMPact FORTRAN multiprocessor enhancement package. To get faster code, use this option on a multiprocessor SPARC system. On a single-processor system the generated code usually runs slower.

Before you use -parallel, see Appendix C, "iMPact: Multiple Processors," Appendix D, "iMPact: Automatic Parallelization," and Appendix E, "iMPact: Explicit Parallelization."

-pentium Generate code for pentium (*x86 only*).

Generate code that exploits features available on x86 Pentium compatible computers. Default: -386. Code compiled with -pentium does run on 80386 and 80486 hardware, but it may run slightly slower.

-pg Profile by procedure for `gprof`.

Produce counting code in the manner of `-p`, but invoke a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. Then you can make an execution profile by running `gprof` (1). `-pg` and `gprof` are complementary to `-a` and `tcov`.

Library options must be *after* the `.f` and `.o` files (`-pg` libraries are static).

If you compile and link in separate steps, and you compile with `-pg`, then be sure to link with `-pg`. Compare this profiling method with the one described in the manual *Performance Tuning an Application*.

For Solaris 2.x, when the operating system is installed, `gprof` is included if you do a *Developer Install*, rather than an *End User Install*; it is also included if you install the package `SUNWbtool`.

-pic Produce position-independent code.

This kind of code is for dynamic shared libraries. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in program-counter-relative addressing mode through a procedure linkage table.

SPARC only, with `-pic`:

- The size of the global offset table is limited to 8K.
- Do not mix `-pic` and `-PIC`.

-PIC Produce position-independent code with 32-bit addresses (*SPARC only*).

Similar to `-pic`, but allows the global offset table to span the range of 32-bit addresses. This is for those rare cases where there are too many global data objects for `-pic`. Do not mix `-pic` and `-PIC`.

-qp Synonym for `-p`.

-Qdir Synonym for `-Qpath`.

-Qoption *prog oplis* Pass option list to program.

Pass the option list *oplis* to the program *prog*. There must be a blank between `Qoption` and *prog* and *oplist*. The `Q` can be uppercase or lowercase. The list is a comma-delimited list of options, no blanks within the list. Each option must be appropriate to that program and can begin with a minus sign. The program can be one of: `as`, `cg`, `cpp`, `fbe`, `f77pass0`, `f77pass1`, `iropt`, `ld`, or `ratfor`.

Solaris 2.x: The assembler used by the compiler is named `fbe`.
Solaris 1.x: The assembler is named `as`.

Example: Pass the linker option `-s` to the linker `ld`.

```
demo$ f77 -Qoption ld -s src.f
```

Example: Load map, 2.x.

Solaris 2.x

```
demo$ f77 -Qoption ld -m src.f
```

Example: Load map, 1.x.

Solaris 1.x

```
demo$ f77 -Qoption ld -M src.f
```

-Qpath *pathname* Compiler component path—insert specified directory.

Insert the directory *pathname* into the compilation search path. There must be a blank between `Qpath` and *pathname*. The `Q` can be uppercase or lowercase. This is for using alternate components of the set of programs that constitute the compiler. This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver (such files as `*crt*.o` and `bb_link.o`).

-Qproduce *typ* Produce code of the specified type.

Produce code of type *typ*. There must be a blank between `Qproduce` and *typ*. The `Q` can be uppercase or lowercase. The *typ* part is one of the following:

- `.o`, object file from the assembler.
- `.s`, assembler source (from `f77pass1` or `cg`)

-r8 REAL*8, interpret REAL as REAL*8.

Interpret REAL as DOUBLE PRECISION, interpret COMPLEX as DOUBLE COMPLEX. For *SPARC*, also interpret DOUBLE PRECISION as quadruple precision and interpret DOUBLE COMPLEX as quadruple complex.

This adjusts declared variables, literal constants, and intrinsic functions. As an intrinsic function example, SQRT is treated as DSQRT.

This sets the default size for REAL to 8, for INTEGER and LOGICAL to 8, and for COMPLEX to 16. For INTEGER and LOGICAL, the compiler allocates 8 bytes, but does 4-byte arithmetic. For *SPARC*, it also sets the default size for DOUBLE PRECISION to 16, and for DOUBLE COMPLEX to 32.

If you specify the size, then the default size is not used. For example, with REAL*_n_ R, INTEGER*_n_ I, LOGICAL*_n_ L, and COMPLEX*_n_ Z, the sizes of R, I, L, and Z are not affected by -r8.

In general, if you compile *one* subprogram with -r8, then be sure to compile *all* subprograms of that program with -r8. Similarly, for programs communicating through files in unformatted I/O, if one program is compiled with -r8, then the other program must also be compiled with -r8.

The impact on runtime performance may be great. Note that with -r8 an expression like "`float = 15.0d0*float`" will be evaluated in quadruple precision due to the declaration of the constant 15.

If you select both –r8 and –i2, the results are unpredictable.

-reduction Reduction loops—analyze loops for reduction (*Solaris 2.x, SPARC only*).

Analyze loops for reduction during automatic parallelization. There is potential for roundoff error with the reduction.

The -reduction option requires the iMPact FORTRAN multiprocessor enhancement package. To get faster code, this option requires a multiprocessor system. On a single-processor system the generated code usually runs slower.

-reduction conflicts with -g.

Before you use -parallel, see Appendix C, "iMPact: Multiple Processors" and Appendix D, "iMPact: Automatic Parallelization."

Reduction works only during automatic parallelization. If you specify `-reduction` without `-autopar`, the compiler does no reduction. If you have a pragma explicitly specifying a loop, then there will be no reduction for that loop.

Example: Automatically parallelize with *reduction*.

```
demo$ f77 -autopar -reduction any.f
```

-R *list* Library paths—store into object file (*Solaris 2.x only*).

While building the executable file, store a list of library search paths into it.

- *list* is a colon-separated list of directories for library search paths.
- The blank between `-R` and *list* is optional.

Multiple instances of this option are concatenated together, with each list being separated by a colon.

How this list is used—The list will be used at runtime by the runtime linker, `ld.so`. At runtime, dynamic libraries in the listed paths are scanned to satisfy any unresolved references.

Why You would want to use it—Use this option to let your users run your executables without a special path option to find your dynamic libraries.

LD_RUN_PATH *and* -R

For `f77`, `-R` and the environment variable `LD_RUN_PATH` are *not* identical (for the runtime linker `ld.so`, they are).

If you build `a.out` with:

- `-R`, then only the paths of `-R` are put in `a.out`. So `-R` is *raw*: it inserts only the paths you name, and no others.
- `LD_RUN_PATH`, then the paths of `LD_RUN_PATH` are put in `a.out`, plus paths for FORTRAN libraries. So `LD_RUN_PATH` is *augmented*: it inserts the ones you name, plus various others.
- Both `LD_RUN_PATH` and `-R`, then only the paths of `-R` are put in `a.out`, and those of `LD_RUN_PATH` are *ignored*.

-s Strip the executable file of its symbol table (makes debugging impossible).

-sb SourceBrowser—compile for the SourceBrowser.

Produce table information for SourceBrowser.

-sbfast SourceBrowser fast—compile fast for the SourceBrowser.

Produce table information for SourceBrowser and stop. Do not call the assembler, linker, and so forth. Do not make object files.

-silent Quiet compile—show prompt only (reduce number of messages from compiler).

If there are no compilation warnings or errors, then display only the prompt. The default is to display the entry names and the file names.

-stackvar Stack the local variables to allow better optimizing with parallelizing.

Use the stack to allocate all *local* variables and arrays in a routine unless otherwise specified. This makes them *automatic*, rather than static.

Purpose: More freedom to optimizer for parallelizing a `CALL` in a loop.

Definition: Variables and arrays are *local* unless they are:

- Arguments in a `SUBROUTINE` or `FUNCTION` statement (on stack anyway)
- Global items in a `COMMON` or `SAVE`, or `STATIC` statement
- Initialized items in a type statement or a `DATA` statement, such as:
  `REAL X/8.0/` or `DATA X/8.0/`

Compilation *errors* can occur with `-stackvar` if a *local* variable is in any of:

- `EQUIVALENCE`
- `NAMELIST`
- `STRUCTURE`
- `RECORD`

Segmentation Fault and `-stackvar`: You can get a segmentation fault using `-stackvar` with *large* arrays. Putting large arrays onto the stack can overflow the stack, so you may need to increase the stack size.

There are two stacks.

- The whole program has a *main* stack.
- Each thread of a multi-threaded program has a *thread* stack.

The default stack size is about 8 Meg for the main stack and 256 K for each thread stack. The `limit` command (no parameters) shows the current main stack size. If you get a segmentation fault using `-stackvar`, you might try doubling the main stack size at least once.

Example: *Stack size*—show the current *main* stack size.

The main stack size →

```
demo$ limit
cputime         unlimited
filesize        unlimited
datasize        523256 kbytes
stacksize       8192 kbytes
coredumpsize    unlimited
descriptors     64
memorysize      unlimited
demo$ ■
```

Example: Set the *main* stack size to 64 Meg.

```
demo% limit stacksize 65536
```

Example: Set each *thread* stack size to 8 Meg.

```
demo% setenv STACKSIZE 8192
```

See `csh`(1) for details on the `limit` command

`-S` Assembly source—generate only assembly source code.

Compile the named programs and leave the assembly-language output on corresponding files suffixed with `.s` (no `.o` file is created).

`-temp=`*`dir`* Temporary files—set directory to locate temporary files.

Set directory for temporary files used by `f77` to be *dir*. No space is allowed within this option string. Without this option, they go into `/tmp/`.

-time Time for execution—display for each compilation pass.

-u Undeclared variables—show a warning message.

Make the default type for all variables be *undeclared* rather than using FORTRAN implicit typing. This warns of undeclared variables. This does not override any `IMPLICIT` statements or explicit *type* statements.

-unroll=*n* Unroll loops—direct the optimizer on unrolling loops *n* times.

Directions to the optimizer about unrolling loops:

- *n* is a positive integer.
- If *n*=1, this *directs* the compiler to unroll *no* loops.
- If *n*>1, this *suggests* to the compiler that it unroll loops *n* times.
- If any loops are actually unrolled, then the executable file is larger.

-U Uppercase identifiers—leave identifiers in the original case.

Do not convert uppercase letters to lowercase, but leave them in the original case. The default is to convert to lowercase except within character-string constants.

If you debug FORTRAN programs that use other languages, it is generally safer to compile with the `-U` option to get case-sensitive variable recognition. If you are not consistent concerning the case of your variables, this `-U` option can cause problems. That is, if you sometimes type `Delta` or `DELTA` or `delta` then the `-U` makes them different symbols.

-v Verbose—print name of each compiler pass.

Print the name of each pass as the compiler executes, plus display in detail the options and environment variables used by the driver.

-vpara Verbose parallelization—show warnings (*Solaris 2.x, SPARC only*).

As the compiler detects each explicitly parallelized loop that has dependencies, it issues a warning message, but the loop is still parallelized.

The `-vpara` option requires the iMPact FORTRAN multiprocessor package.

Use `-vpara` with the `-explicitpar` option and the "`c$par doall`" pragma.

Example: `-vpara` for verbose parallelization warnings.

```
demo$ f77 -explicitpar -vpara any.f
```

`-V` Version ID—]show version ID.

Print the name and version ID of each pass as the compiler executes.

`-w` Warnings, suppress—do no show warnings.

Suppress warning messages. This suppresses most warnings. However, if one option overrides all or part of an option earlier on the command line, you do get a warning.

Example: `-w` still allows some warnings to get through.

```
demo$ f77  -w -fast  -silent -O4  any.f
f77: Warning: -O4 overwrites previously set optimization level of -O3
demo$ ■
```

`-xa` Synonym for `-a`.

`-xcg89` Synonym for `-cg89`.

`-xcg92` Synonym for `-cg92`.

`-xF` Reorder functions—enable reordering of functions in core (*Solaris 2.x only*).

Enable the reordering of functions in the core image using the compiler, the Analyzer and the linker. If you compile with the `-xF` option, then run the Analyzer, you get a map file that shows an optimized order for the functions. The subsequent link to build the executable file can be directed to use that map by using the linker `-Mmapfile` option. Within the mapfile, if you include the flag "`O`" (that's an *oh*, for *order*, not a *zero*) in the string of segment flags, then the static linker `ld` will attempt to place sections in the order they appear in the mapfile.

Example: -xF, In the mapfile, there can be a line such as this:

```
text = LOAD ? RXO
```

See the analyzer(1) and debugger(1) man pages.

-xinline=*rlst* Synonym for -inline=*rlst*.

-xl[d] Extend the language, VMS FORTRAN.

-xl: Extend the language, VMS.

Although you get most VMS features automatically, without any special options, for a few VMS features you must use the -xl option.

In general you need the -xl option if a source statement can be interpreted as either a VMS feature or an f77 feature, and you want the VMS feature. In this case the -xl option forces the compiler to interpret it the VMS way.

The following VMS language features require the -xl[d] option.

- Unformatted record size in words rather than bytes (-xl)
- VMS style logical file names (-xl)
- Quote (") character introducing octal constants (-xl)
- Backslash (\ ) as ordinary character within character constants (-xl)
- Nonstandard form of the PARAMETER statement (-xl)
- Debugging lines as comment lines or FORTRAN statements (-xld)
- Align structures as in VMS. (-xl)

  VMS Alignment—Use the -xl to get VMS alignment if your program has some detailed knowledge of how VMS structures are implemented.

  If you use both -oldstruct and -xl, then you get -oldstruct. If you need to share structures with C, use the default; do not use -xl and do not use -oldstruct.

You may also be interested in -lV77 and the VMS library. See Section 7.5, “Libraries Provided,” on page 135.

Read the chapter on VMS language extensions in the *FORTRAN Reference Manual* for details of the VMS features that you get automatically.

-xld Debug comments—extended language, VMS, plus *debug* comments

In addition to the features you get with `-xl`, the `-xld` option causes debugging comments (`D` or `d` in column one) to be compiled. Without the `-xld` option, they remain comments only. No space is allowed between `-xl` and `d`.

-xlibmil Synonym for `-libmil`.

-xlibmopt (SPARC only) Use selected math routines optimized for performance.

This usually generates faster code. It may produce slightly different results; if so, they usually differ in the last bit. The order on the command line for this library option is not significant.

-xlicinfo License information—display license server user ids.

Return license information about the licensing system. In particular, return the name of the license server and the user ID for each of the users who have licenses checked out. Generally, with this option no compilation is done and a license is not checked out; and generally this option is used with no other options. However, if a conflicting option is used, then the last one on the command line wins and there is a warning.

Example: Report license information, do not compile (order counts).

```
demo$ f77  -c -xlicinfo any.f
```

Example: Do not report license information, do compile (order counts).

```
demo$ f77  -xlicinfo -c any.f
```

-xnolib Synonym for `-nolib`.

-xnolibmil Synonym for `-nolibmil`.

-xnolibmopt (SPARC only) Reset `-fast` so that it does not use `-xlibmopt`.

Reset `-fast` so that it does not use the library of selected math routines optimized for performance. Use this after the `-fast` option:
`f77 -fast -xnolibmopt ...`

-xO[n] Synonym for -O[*n*].

-xpg Synonym for -pg.

-xs Autoload off—disable *autoload* for dbx (*Solaris 2.x only*).

Use this is in case you cannot keep the .o files around. This passes -s to the assembler and linker.

No Autoload: This is the older way of loading symbol tables.

- Place all symbol tables for dbx in the executable file.
- The linker links more slowly and dbx initializes more slowly.

  If you move executables to another directory, then to use dbx you *need not* move the object (.o) files.

Autoload: This is the newer (and default) way of loading symbol tables.

- Distribute this information in the .o files so that dbx loads the symbol table information only if and when it is needed.
- The linker links faster and dbx initializes faster.

  If you move the executables to another directory, then to use dbx you *must* move the object (.o) files.

-xsb Synonym for -sb.

-xsbfast Synonym for -sbfast.

-xtime Synonym for -time.

-xunroll=*n* Synonym for -unroll=*n*.

-Xlist Global program checking—check across routines (arguments, commons, …).

This helps find a variety of bugs by checking across routines for consistency in arguments, common blocks, parameters, and so forth. In general, `-Xlist` also makes a line-numbered listing of the source, and a cross reference table of the identifiers. The errors found do not prevent the program from being compiled and linked.

Example: Check across routines for consistency.

```
demo$ f77 -Xlist  fil.f
```

The above example shows the following in the output file `fil.lst`:

- A line-numbered source listing (Default)
- Error messages (embedded in the listing) for inconsistencies across routines
- A cross reference table of the identifiers (Default)

See “Global Program Checking (-Xlist)” on page 139,” for details.

-Zlp Loops—profile all the loops for MP (*Solaris 2.x, SPARC only*).

Prepare object files for the loop profiler, `looptool`. The `looptool`(1) utility can then be run to generate loop statistics about the program.

The `-Zlp` option requires the iMPact FORTRAN multiprocessor package.

If you compile and link in separate steps, and you compile with `-Zlp`, then be sure to *link* with `-Zlp`.

If you compile *one* subprogram with `-Zlp`, you need not compile *all* subprograms of that program with `-Zlp`. However, you get loop information only for the files compiled with `-Zlp`, and no indication that the program includes other files.

Refer to the *Thread Analyzer User’s Guide* for more information.

-Ztha Prepare object files for the Thread Analyzer (*Solaris 2.x, SPARC only*).

It inserts calls to a profiling library at all procedure entries and exits. Code compiled with `-Ztha` links with the library `libtha.so`. The `-Ztha` option requires the iMPact FORTRAN MP package.

If you compile and link in separate steps, and you compile with `-Ztha`, then link with `-Ztha`.

If you compile a subprogram with `-Ztha`, you need not compile all subprograms of that program with `-Ztha`. However, you get thread statistics only for the files compiled with `-Ztha`, and no indication that the program includes other files.

Refer to `tha` (1), or the *Thread Analyzer User's Guide* for more information.

3.6 Native Language Support

This version of FORTRAN supports the development of applications using languages other than English, including most European languages. As a result, you can switch the development of applications from one native language to another.

This FORTRAN implements internationalization as follows:

- It recognizes 8-bit characters from European keyboards supported by Sun.
- It allows the printing of your own messages in the native language. (It is 8-bit clean.)
- It allows native language characters in comments, strings, and data.
- It allows you to localize the compile-time error messages files.

Locale

You can enable changing your application from one native language to another by setting the locale. This changes some aspects of displays, such as date and time formats. For information on this and other native language support features, read Chapter 6, "Native Language Application Support," of the System Services Overview for Solaris software.

Some Aspects Cannot Change

Even though some aspects can change if you set the locale, certain aspects cannot change. An internationalized SPARCompiler language does not allow input and output in the various international formats. If it did, it would not comply with the language standard appropriate for its language. For example, some languages have standards that specify a period (.) as the decimal unit in the floating-point representation.

Example: No I/O in international formats.

native.f

```
      PROGRAM sample
      REAL r
      r = 1.2
      WRITE( 6, 1 ) r
1     FORMAT( 1X F10.5 )
      END
```

Here is the output.

```
    1.20000
```

In the example above, if you reset your system locale to, say, France, and rerun the program, you'll still get the same output. The period won't be replaced with a comma, the French decimal unit.

Compile-Time Error Messages

The compile-time error messages are on files called source catalogs so you can edit them. You may decide to translate them to a local language such as French or Italian. Usually a third party does the translating. Then you can make the results available to all local users of `f77`. Each user of `f77` can choose to use these files or not.

Localizing and Installing the Files

Usually a system administrator does the installing. It is generally done only once per language for the whole system, rather than once for each user of `f77`. The results are available to all users of `f77` on the system.

1. Find the source catalogs.

The file names are:

- `SUNWspro_f77pass1_srccat` (about 300 error messages)
- `SUNWspro_compile_srccat` (about 10 error messages)

2. Edit the source catalogs.

a. Make backup copies of the files.

b. In whatever editor you are comfortable with, edit the files.

This can be `vi`, `emacs`, `textedit`, and so forth.

Preserve any existing format directives, such as `%f`, `%d`, `%s`, and so forth.

c. Save the files.

3. Generate the binary catalogs from the source catalogs.

The compiler uses only binary catalogs. Run the `gencat` program twice.

a. Read the `SUNWspro_f77pass1_srccat` source file and generate the `SUNWspro_f77pass1_cat` binary file.

```
demo$ gencat SUNWspro_f77pass1_cat  SUNWspro_f77pass1_srccat
```

b. Read the `SUNWspro_compile_srccat` source file and generate the `SUNWspro_compile_cat` binary file.

```
demo$ gencat SUNWspro_compile_cat  SUNWspro_compile_srccat
```

4. Make the binaries available to the general user.

Either put the binary catalogs into the standard location or put the path for them into the environment variable NLSPATH.

a. Standard location and name

Put the files into the directory indicated:

Solaris 2.x

```
/opt/SUNWspro/lib/locale/lang/LC_MESSAGES/
```

Solaris 1.x

```
/usr/share/lib/locale/lang/LC_MESSAGES/
```

where *lang* is the directory for the particular (natural) language. For example, the value of *lang* for Italian is *it*.

b. Environment variable

Put the path for the new files into the NLSPATH environment variable. Example: If your files are in /usr/local/MyMessDir/ then use the commands shown below.
sh:

```
demo$ NLSPATH=/usr/local/MyMessDir
demo$ export NLSPATH
```

csh:

```
demo% setenv NLSPATH  /usr/local/MyMessDir
```

The NLSPATH variable is standard for the X/Open environment. For more information, read the X/Open documents.

Using the File After It Is Installed

You use the file by setting the environment variable LC_MESSAGES. This is generally done once for each developer.

Example: Set the environment variable LC_MESSAGES.
sh:

This example assumes:
1. Standard install locations
2. Messages localized in Italian

```
demo$ LC_MESSAGES it
demo$ export it
```

csh:

```
demo% setenv  LC_MESSAGES it
```

3.7 Compiler Directives

A pragma passes information to a compiler in a special form of comment. ♦

Pragmas are also called *compiler directives.* There are two kinds of pragmas:

- General pragmas
- Parallel pragmas

General Pragmas

There is one general pragma, the C pragma. The form of a pragma is:

```
c$pragma id
or
c$pragma id ( a [ , a ] … ) [ , id ( a [ , a ] … ) ] ,…
```

The variable *id* identifies the kind of pragma, and *a* is an argument.

Syntax

A general pragma has the following syntax.

- In column one, any of the comment-indicator characters `c`, `C`, `d`, `D`, `!`, or `*`
- In *any* column, the `!` comment-indicator character
- The next 7 characters are `$pragma`, no blanks, any uppercase/lowercase.

Rules and Restrictions for General Pragmas

- After the first eight characters, blanks are ignored, and uppercase and lowercase are equivalent, as in FORTRAN text.
- Because it is a comment, a pragma cannot be continued, but you can have many `c$pragma` lines, one after the other, as needed.
- If a comment satisfies the above syntax, it is expected to contain one or more directives recognized by the compiler; if it does not, a warning is issued.

The C Directive

The `C()` directive specifies that its arguments are external functions written in the C language. It is equivalent to an `EXTERNAL` declaration with the addition that the FORTRAN compiler does not append an underscore to such names, as it ordinarily does with external names. See Chapter 13, “C–FORTRAN Interface” for more detail.

The C() directive for a particular function must appear before the first reference to that function in each subprogram that contains such a reference.

Example: To compile SUBx and FCNy for C.

```
      EXTERNAL ABC, XYZ  !$PRAGMA C(ABC, XYZ)
```

Parallel Pragmas

A *parallel pragma* directs the compiler to do some parallelizing. The syntax of parallel pragmas is different from the syntax of general pragmas.

Syntax

A *parallel* pragma has the following syntax.

- The first character must be in column one.
- The first character can be any one of c, C, d, D, *, or !.
- The next 4 characters are $par, no blanks, any uppercase and lowercase.

A parallel pragma differs slightly from the more general pragma in the following ways:

- A parallel pragma must start in column one.
- The initial characters are $par as in c$par or *$par, ...

Usage

Currently there is only one parallel pragma, the doall pragma.

- c$par doall

For doall, the compiler parallelizes the *next loop* it finds after the pragma, if possible. Before using parallel pragmas, see Appendix E, "iMPact: Explicit Parallelization."

3.8 Miscellaneous Tips

Floating-Point Hardware Type

Some compiler options are specific to particular hardware options. The utility `fpversion` tells which floating-point hardware is installed. The utility `fpversion`(1) takes 30 to 60 wall clock seconds before it returns, since it dynamically calculates hardware clock rates of the CPU and FPU. See `fpversion`(1). Also read the *Numerical Computation Guide* for details.

Many Options on Short Commands

Some users type long command lines—with many options. To avoid this, make a special alias or use environment variables.

Alias Method

Example: Define `f77f`.

```
demo$ alias f77f "f77 -silent -fast -O4"
```

Example: Use `f77f`.

```
demo$ f77f any.f
```

The above command is equivalent to "`f77 -silent -fast -O4 any.f`"

Environment Variable Method

Some users shorten command lines by using environment variables. The `FFLAGS` or OPTIONS variables are a special variables for FORTRAN.

- If you set `FFLAGS` or `OPTIONS`, they can be used in the command line.
- If you are compiling with `make` files, `FFLAGS` is used automatically if the `make` file uses only the implicit compilation rules.

Example: Set `FFLAGS`.

```
  demo$ setenv FFLAGS '-silent -fast -O4'
```

- Example: Use FFLAGS explicitly.

```
demo$ f77 $FFLAGS any.f
```

The above command is equivalent to "f77 -silent -fast -O4 any.f"

- Example: Let make use FFLAGS implicitly.

If both:

- The compile in a make file is *implicit* (*no explicit* f77 compile line)
- The FFLAGS variable is set as above

Then invoking the make file results in a compile command equivalent to "f77 -silent -fast -O4 any.f"

Align Block

In Solaris 2.x the –align _*block*_ option is available only for compatibility with old make files. It is recognized, so it does not break such files, but it does not do anything. However, you can still page-align a common block.

This applies to *uninitialized* data only. If any variable of the common block is initialized in a DATA statement, then the block will not be aligned. This aligns the common block on a page boundary. Its size is increased to a whole number of pages and its first byte is placed at the beginning of a page.

Example: Page-align the common block whose FORTRAN name is *block*.

```
      COMMON /BLOCK/ A, B
      REAL*4 A(11284), B(11284)
      ...
```

This block has a size of 90,272 bytes. You must create a separate assembler source file (.s file) consisting of the following .common statement.

```
demo$ cat comblk.s
.common block_,90272,4096
demo$ ■
```

For this example:

- block is the FORTRAN name of the block. Note appended underscore (_).
- 90272 is the block size in bytes.
- 4096 is the page size in bytes. (Some systems have different page sizes.)

If you do *not* use the `-U` option, use lowercase for the common block name. If you *do* use `-U`, use the case of the common block name in your source code. The parameters are *block name*, *block size*, and *page size*.

You must compile and link the `.s` file with the `.f` file(s).

```
demo$ f77 any.f comblk.s
```

The stricter alignment from this file should override the less strict alignment for the common block from the other `.o` files.

Optimizer Out of Memory

Optimizers use a lot of memory. For SPARC systems, if the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent routines at the original level specified in the `-O`*n* option on the command line.

It is recommended that you have at least 24 megabytes of memory. If you do full optimization, at least 32 megabytes is recommended. How much you need depends on the size of each procedure, the level of optimization, the limits set for virtual memory, the size of the disk swap file, and certainly various other parameters.

If the optimizer runs out of swap space, try any of the following measures (listed in increasing order of difficulty).

- Change from `–O3` to `–O2`.
- Use `fsplit` to divide multiple-routine files into files of one routine per file.
- Allow space for a bigger swap file. See `mkfile`(8).

Example (*2.x*): Be superuser, make a 90 megabyte file, tell system to use it.

```
demo# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
demo# /usr/sbin/swap -a  /home/swapfile
```

Example (*1.x*): Be superuser, make the file, and tell the system to use it.

```
demo# mkfile -v 20m /home/swapfile
/home/swapfile 20971520 bytes
demo# swapon  /home/swapfile
```

The swap command, above, must be reissued every time you reboot, or added to the appropriate `/etc/rc file`. The Solaris 2.x command "`swap -s`" displays available swap space. See `swap`(1M).

- Control the amount of virtual memory available to a single process.

  If you optimize at `-O3` or `-O4` with very large routines (thousands of lines of code in a single procedure), the optimizer may require an unreasonable amount of memory. In such cases, performance of the machine may be degraded. You can control this by limiting the amount of virtual memory available to a single process.

Virtual Memory Limits

Limit virtual memory.

- In `sh`, use the `ulimit` command (see `sh`(1)).

  Example: To limit virtual memory to 16 megabytes:

```
demo$ ulimit -d 16000
```

- In `csh`, use the `limit` command (see `csh`(1)).

  Example: To limit virtual memory to 16 megabytes:

```
demo% limit datasize 16M
```

Each causes the optimizer to try to recover at 16 megabytes of data space.

This limit cannot be greater than the machine's total available swap space and, in practice, must be small enough to permit normal use of the machine while a large compilation is in progress.

Ensure that no compilation consumes more than half the space.

Example: On a machine with 32 megabytes of swap space, use the commands shown below.

- In `sh`:

```
demo$ ulimit -d 1600
```

- In `csh`:

```
demo% limit datasize 16M
```

The best setting of data size depends on the degree of optimization requested and the amount of real memory and virtual memory available.

Swap Space Limits

Find the actual swap space (either `sh` or `csh`).

Example: Actual swap space—use the `swap` command in 2.x.

Solaris 2.x
```
demo$ swap -s
```

Example: Actual swap space—use the `pstat` command in 1.x.

Solaris 1.x
```
demo$ pstat -s
```

Memory

Find actual real memory, (either `sh` or `csh`).

- Example: Use the following command in 2.x.

Solaris 2.x
```
demo$ /usr/sbin/dmesg | grep mem
```

- Use either of the following commands in 1.x.

Solaris 1.x
```
demo$ /etc/dmesg | grep mem
```

or

Solaris 1.x
```
demo$ grep mem /var/adm/messages*
```

3

BCP Mode: How to Make 1.x Applications Under 2.x

This section shows some details of how, in a Solaris 2.x operating environment, to compile and link applications that run in a Solaris 1.x operating environment.

Note – This is not recommended. It is possible to produce 1.x executables on a 2.x development platform, but most developers consider it too complicated.

Read SunSoft's "*Solaris 2.3 Binary Compatibility Manual*" first.

The usual way of operating is as follows:

- The 1.x compilers in `/usr/lang/` are used on 1.x platforms to produce 1.x executables.
- The 2.x compilers in `/opt/SUNWspro/bin` are used on 2.x platforms to produce 2.x executables.

To use a 2.x operating environment to make executables that run under 1.x, the following steps are necessary:

- Be sure the appropriate BCP packages are installed:
  - `SUNWbcp`: Binary Compatibility
  - `SUNWscbcp`: SPARCompilers Binary Compatibility

  Use `pkginfo` to verify this. The binary compatibility libraries are installed in `/usr/4lib`.
- Be sure to use the 1.x compiler, not the native 2.x compiler.

  This may require installing it in a nonstandard location to make it accessible on the 2.x platform.
- If possible, perform the final link of object files on a 1.x platform.

  Otherwise, in order to link your 1.x executable successfully on the 2.x platform, you need access to a 1.x version of ld.
  - Copy a 1.x version of `ld` to *1.x_ld_path*`/ld`.
  - Tell `f77` where to find the 1.x linker by supplying its path via `-Qpath`.

- Make versions of the 1.x libraries available on 2.x

  On 1.x, copy files from `/lib` to *com_dir*.

```
demo$ cp -p /lib/libc.a com_dir
demo$ cp -p /lib/libc.so.1.8 com_dir
demo$ cp -p /lib/libc.sa.1.8 com_dir
                Plus any other system 1.x libraries you need
demo$
```

  On 2.x, break the link `/lib -> /usr/lib`.

```
demo$ su root
Password: your_own_root_pasword
#mv /lib /lib-                          Rename the link /lib to /lib-
#mkdir /lib                             Make a new directory /lib
#demo$
```

  On 2.x, copy the same files from *com_dir* to `/lib`.

```
#mv com_dir/libc.a /lib  Move the same files from com_dir to /lib
#mv com_dir/libc.so.1.8 /lib
#mv com_dir/libc.sa.1.8 /lib
                Plus any other system 1.x libraries you need
#exit
demo$
```

  Or, you can move the needed libraries into a directory of your choosing and enable the linker to find these libraries by supplying the path via the `-L` option or the `LD_LIBRARY_PATH` environment variable.

  Make sure 1.x libraries that are non-system libraries are available, say in the *1.x_lib_path dir.*

- Compile the program:

```
demo$ 1.x_f77_path/f77 -Qpath 1.x_ld_path -L1.x_lib_path file.f
```

  Assumes:

  - `f77` is in *1.x_f77_path/*
  - `ld` is in *1.x_ld_path*
  - libraries are in *1.x_lib_path*

- Pitfalls
  - If `libc` is linked statically then all libraries must be linked statically.
  - `/usr/4lib` is not a suitable choice for *1.x_lib_path* or for system libraries you copy, because the presence of `libc.so.101.8` and `libc.so.102.8` frustrates linking.
  - The message "`bad magic number`" probably means that you attempted to link with a 2.x library instead of a 1.x library. Check your command line for inappropriate `-L` options; check `LD_LIBRARY_PATH`. Remember that the 1.x linker searches the following paths for libraries:

Environment variable	`LD_LIBRARY_PATH`
Directories specified at link-time	`-L`*dir*
Default directories	`/lib:/usr/lib:/usr/local/lib`

  - Normally `LD_LIBRARY_PATH` will point to 2.x libraries on a 2.x platform; it may need to be reset.

    Especially, do not put `/usr/lib` in `LD_LIBRARY_PATH`.

- Summary
  - The resultant `a.out` should run on 1.x systems. It may run (in BCP mode) on 2.x systems including the development platform; check the "*Solaris 2.3 Binary Compatibility Manual*" for guidelines.
  - You may want to replace your 1.x shared libraries in 1.*x_lib_path* with 2.x libraries, since `a.out` will try to link with them.
  - Use `ldd` to find which shared libraries `a.out` links with.
  - On a 1.x system your `a.out` will look for shared libraries in the directories that were found on your development platform; differences in directory structure may cause problems.

File System and FORTRAN I/O 4

This chapter is organized into the following sections.

Summary	*page 77*
Directories	*page 79*
File Names	*page 79*
Path Names	*page 79*
Redirection	*page 82*
Piping	*page 83*

This chapter is a basic introduction to the file system and how it relates to the FORTRAN I/O system. If you understand these concepts, then skip this chapter.

4.1 Summary

The basic file system consists of a hierarchical file structure, established rules for file names and path names, and various commands for moving around in the file system, showing your current location in the file system, and making, deleting, or moving files or directories.

The system file structure of the UNIX operating system is analogous to an upside-down tree. The top of the file system is the *root* directory. Directories, subdirectories, and files all branch *down* from the root. Directories and subdirectories are considered nodes on the directory tree, and can have

subdirectories or ordinary files branching down from them. The only directory that is not a subdirectory is the root directory, so except for this instance, you do not usually make a distinction between directories and subdirectories.

A sequence of branching directory names and a file name in the file system tree describes a *path*. Files are at the ends of paths, and cannot have anything branching from them. When moving around in the file system, *down* means away from the root and *up* means toward the root. The figure below shows a diagram of a file system tree structure.

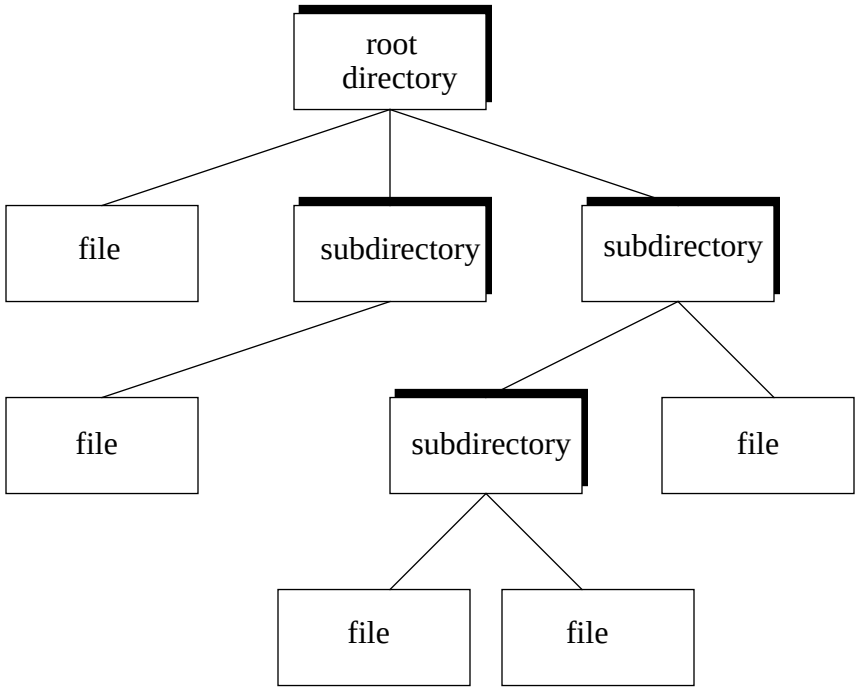


Figure 4-1 File System Hierarchy

4.2 Directories

All files branch from directories except the root directory. Directories are just files with special properties. While you are logged on, you are said to be *in a directory*. When you first log on, you are usually in your *home* directory. At any time, wherever you are, the directory you are in is called your *current working directory*. It is often useful to list your current working directory. The `pwd` command *prints* the current working directory name and the `getcwd` routine *gets* (returns) the current working directory name. You can change your current working directory simply by moving to another directory. The `cd` shell command and the chdir routine change the current working directory to a different directory.

4.3 File Names

All files have names, and you can use almost any character in a file name. The name can be up to 1024 characters long, but individual components can be only 512 characters long. However, to prevent the shell from misinterpreting certain special punctuation characters, restrict your use of punctuation in file names to the dot (.), underscore (_), comma (,), plus (+), and minus (-). The slash (/) character has a specific meaning in a file name, and is only used to separate components of the path name (as described in the following section). Also, avoid using blanks in file names. Directories are just files with special properties and follow the same naming rules as files. The only exception is the root directory, named slash (/).

4.4 Path Names

To describe a file anywhere in the file system, you can list the sequence of names for the directory, subdirectory, and so forth, and file, separated by slash characters, down to the file you want to describe. If you show *all* the directories, starting at the *root*, that's called an *absolute* path name. If you show only the directories below the current directory, that's called a *relative* path name.

Relative Path Names

From anywhere in the directory structure, you can describe a *relative path name* of a file. Relative path names start with the directory you are in (the current directory) instead of the root. For example, if you are in the directory `/usr/you`, and you use the *relative* path name `mail/record`, that is equivalent to using the *absolute* path name `/usr/you/mail/record`.

This is illustrated in the diagram below:

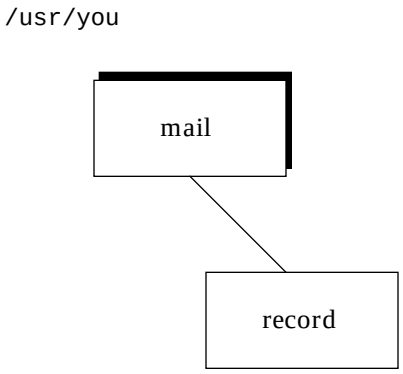


Figure 4-2 Relative Path Name

Absolute Path Names

A list of directories and a file name, separated by slash characters, from the root to the file you want to describe, is called an *absolute path name*. It is also called the *complete file specification* or the *complete path name*.

A complete file specification has the general form:

```
/directory/directory/…/directory/file
```

There can be any number of directory names between the root (/) and the file at the end of the path as long as the total number of characters in a given path name is less than or equal to 1024.

An absolute path name is illustrated in the diagram below:

```
/usr/you/mail/record
```

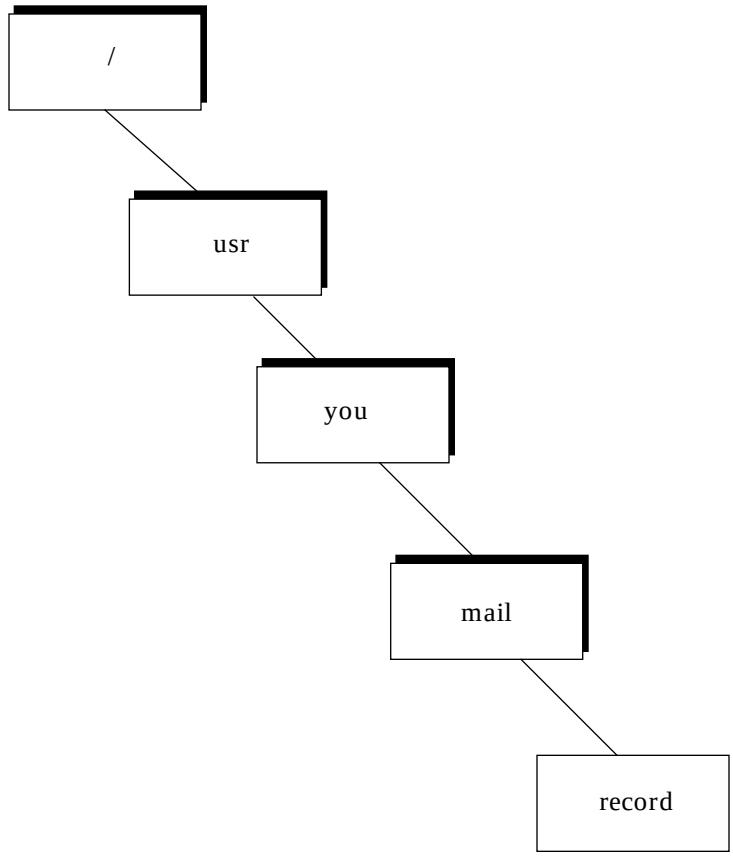


Figure 4-3 Absolute Path Name

4.5 Redirection

Redirection is a way of changing the files that a program uses without passing a file name to the program. Both input to and output from a program can be redirected. The usual symbol for redirecting standard input is the '<' sign, and for standard output is the ">" sign. File redirection is a function performed by the command interpreter or *shell* when a program is invoked by it.

Input

The shell command line for `myprog` to *read* from `mydata` is as follows:

```
demo$ myprog < mydata
```

The above command causes the file `mydata` (which must already exist) to be connected to the standard input of the program `myprog` when it is run. This means that if `myprog` is a FORTRAN program and reads from unit 5, it reads from the `mydata` file.

Output/Truncate

The shell command line for `myprog` to *write* to `myoutput` is as follows:

```
demo$ myprog > myoutput
```

The above command causes the file `myoutput` (which is created if it does not exist, or rewound and truncated if it does) to be connected to the standard output of the program `myprog` when it is run. So if the FORTRAN program `myprog` writes to unit 6, it writes to the file `myoutput`.

Output/Append

The shell command line for `myprog` to *append* to `mydata` is as follows:

```
demo$ myprog >> myoutput
```

The above command causes the file `myoutput` (which must exist) to be connected for *appending*. So if the FORTRAN program `myprog` writes to unit 6, it writes to the file `myoutput` but after wherever the file ended before.

You can redirect standard input and output on the same command line.

4.6 Piping

You can connect the standard output of one program directly to the standard input of another without using an intervening temporary file. The mechanism to accomplish this is called a *pipe*. Some consider piping to be a special kind of redirecting.

Example: A shell command line using a pipe.

```
demo$ firstprog | secondprog
```

This causes the standard output (unit 6) of `firstprog` to be piped to the standard input (unit 5) of `secondprog`. Piping and file redirection can be combined in the same command line.

Example: `myprog` reads `mydata` and pipes output to `wc`, `wc` writes `datacnt`.

```
demo$ myprog < mydata | wc > datacnt
```

The program `myprog` takes its standard input from the file `mydata`, and has its standard output piped into the standard input of the `wc` command, the standard output of `wc` is redirected into the file `datacnt`.

Standard Error

Standard error may also be redirected so it does not appear on your workstation display. In general, this is not a good idea, since you usually want to get error messages from the program immediately, rather than sending them to a file.

The shell syntax to redirect standard error varies, depending on whether you are using `sh` or `csh`.

Example: `csh`. Redirecting / piping standard error and standard output.

```
demo% myprog1 |& myprog2
```

Example: `sh`. Redirecting / piping standard error and standard output.

```
demo$ myprog1 2>&1 | myprog2
```

In each shell, the above command runs the program `myprog1` and redirects / pipes standard output and error to the program `myprog2`.

4

Disk and Tape Files 5

This chapter is organized into the following sections.

Accessing Files from FORTRAN Programs	*page 85*
Tape I/O	*page 95*

5.1 Accessing Files from FORTRAN Programs

Data are transferred to or from devices or files by specifying a logical unit number in an I/O statement. Logical unit numbers can be nonnegative integers or the character "*". The "*" stands for the *standard input* if it appears in a `READ` statement, or the *standard output* if it appears in a `WRITE` or `PRINT` statement. Standard input and standard output are explained in the section on preconnected units found later in this chapter.

Accessing Named Files

Before a program can access a file with a `READ`, `WRITE`, or `PRINT` statement, the file must be created and a connection established for communication between the program and the file. The file can already exist or be created at the time the program executes. The FORTRAN `OPEN` statement establishes a connection between the program and file to be accessed. (For a description of `OPEN`, read the chapter on statements in the *FORTRAN Reference Manual*.)

File names can be simple expressions such as:

- Quoted character constants:

```
      FILE='myfile.out'
```

- Character variables:

```
      FILE=FILNAM
```

File names can be more complicated expressions such as character expressions:

```
      FILE=PREFIX(:LNBLNK(PREFIX)) // '/' //
&           NAME(:LNBLNK (NAME)),…
```

Some ways a program can get file names are:

- By reading from a file or terminal keyboard, such as:

```
      READ( 4, 401) FILNAM
```

- From the command line, such as:

```
      CALL GETARG( ARGNUMBER, FILNAM )
```

- From the environment, such as:

```
      CALL GETENV( STRING, FILNAM )
```

The example below shows one way to construct a file name in the C shell:

GetFilNam.f

```
      CHARACTER F*8, FN*40, FULLNAME*40
      READ *, F
      FN = FULLNAME( F )
      PRINT *, FN
      END

      CHARACTER*40 FUNCTION FULLNAME( NAME )
      CHARACTER NAME*(*), PREFIX*40
C         This assumes C shell.
C         Leave absolute path names unchanged.
C         If name starts with '~/', replace tilde with home
C         directory; otherwise prefix relative path name with
C         path to current directory.
      IF ( NAME(1:1) .EQ. '/' ) THEN
           FULLNAME = NAME
      ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
           CALL GETENV( 'HOME', PREFIX )
           FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&                       NAME(2:LNBLNK(NAME))
      ELSE
           CALL GETCWD( PREFIX )
           FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&                       '/' // NAME(:LNBLNK(NAME))
      ENDIF
      RETURN
      END
```

Note: This uses the library routines `getenv`, `lnblnk`, and `getcwd`. (get environment, last nonblank, get current working directory)

Compile and run `GetFilNam.f`.

```
demo$ f77 -silent GetFilNam.f
demo$ a.out
"/hokey"
/hokey
demo$
```

Accessing Unnamed Files

When a program opens a FORTRAN file without a name, the runtime system supplies a file name. There are several ways it can do this.

Opened as Scratch

If you specify `STATUS='SCRATCH'` in the `OPEN` statement, then the system opens a file with a name of the form `tmp.F`*`AAAxnnnnn`*, where *nnnnn* is replaced by the current process ID, *AAA* is a string of three characters, and *x* is a letter; the *AAA* and *x* make the file name unique. This file is deleted upon termination of the program or execution of a `CLOSE` statement, unless `STATUS='KEEP'` is specified in the `CLOSE` statement.

Already Open

If a FORTRAN program has a file already open, an `OPEN` statement that specifies only the file's logical unit number and the parameters to change can be used to change some of the file's parameters (specifically, `BLANK` and `FORM`). The system determines that it must not really `OPEN` a new file, but just change the parameter values. Thus, this looks like a case where the runtime system would make up a name, but it does not.

Other

In all other cases, the runtime system `OPEN`s a file with a name of the form `fort.n`, where *n* is the logical unit number given in the `OPEN` statement.

Passing File Names to Programs

The file system does not have any notion of temporary file name binding (or file equating) as some other systems do. File name binding is the facility that is often used to associate a FORTRAN logical unit number with a physical file without changing the program. This mechanism evolved to communicate file names more easily to the running program, because in FORTRAN 66 you can not open files by name.

With this operating system there are several satisfactory ways to communicate file names to a FORTRAN program.

- Command-line arguments and environment-variable values. For example, read the file `ioinit.f` in `libF77`. See the subsection “Logical Unit Preattachment.” The program can then use those logical names to open the files.
- Redirection and piping. The previous chapter describes *redirection* and *piping*, two other ways to change the program input and output files without changing the program.

Preconnected Units

When a FORTRAN program begins execution under this operating system, there are usually three units already open. They are *preconnected units*. Their names are *standard input*, *standard output*, and *standard error*. In FORTRAN:

- Standard input is logical unit 5
- Standard output is logical unit 6
- Standard error is logical unit 0

All three are connected, unless file redirection or piping is done.

Other Units

All other units are preconnected to files named `fort.`*n* where *n* is the corresponding unit number, and can be 0, 1, 2, …, with 0, 5, and 6 having the usual special meanings. These files need not exist, and are created only if the units are actually used, and if the first action to the unit is a `WRITE` or `PRINT`; that is, only if an `OPEN` statement does not override the preconnected name before any `WRITE` or `PRINT` is issued for that unit.

Example: Preconnected Files. The program `OtherUnit.f`.

```
      WRITE( 25, '(I4)' ) 2
      END
```

The above program preconnects the file `fort.25` and writes a single formatted record onto that file.

```
demo$ f77 -silent OtherUnit.f
demo$ a.out
demo$ cat fort.25
   2
demo$
```

Logical Unit Preattachment

The `IOINIT` routine can also be used to attach logical units to specific files at runtime. It looks in the environment for names of a user-specified form, and then it opens the corresponding logical unit for sequential formatted I/O. Names must be of the general form *PREFIXnn*, where the particular *PREFIX* is specified in the call to `IOINIT`, and *nn* is the logical unit to be opened. Unit numbers less than 10 must include the leading "0". See `IOINIT`(3F).

Example: Attach external files `test.inp` and `test.out` to units 1 and 2.

First set environment variables.

`sh`

```
demo$ TST01=ini1.inp
demo$ TST02=ini1.out
demo$ export TST01 TST02
```

`csh`

```
demo% setenv TST01 ini1.inp
demo% setenv TST02 ini1.out
```

The program `ini1.f` reads 1 and writes 2.

```
demo$ cat ini1.f
      CHARACTER PRFX*8
      LOGICAL CCTL, BZRO, APND, VRBOSE
      DATA CCTL, BZRO, APND, PRFX, VRBOSE
&               /.TRUE.,.FALSE.,.FALSE., 'TST',.FALSE. /
      CALL IOINIT( CCTL, BZRO, APND, PRFX, VRBOSE )
      READ(1, *) I, B, N
      WRITE(2, *) I, B, N
      END
demo$ ■
```

With environment variables and `ioinit`, `ini1.f` reads `ini1.inp` and writes `ini1.out`.

```
demo$ cat ini1.inp
 12 3.14159012 6
demo$ f77 -silent ini1.f
demo$ a.out
demo$ cat ini1.out
  12    3.14159  6
demo$ ■
```

`IOINIT` is adequate for most programs as written. However, it is written in FORTRAN specifically so that it may serve as an example for similar user-supplied routines. Retrieve a copy as follows.

```
demo$ cp /opt/SUNWspro/SC3.0.1/src/ioinit.f  .  {Solaris 2.x}
```

Logical File Names

If you are porting from VMS FORTRAN, the VMS style logical file names in the `INCLUDE` statement are mapped to UNIX path names. The environment variable `LOGICALNAMEMAPPING` defines the mapping between the logical names and the UNIX path name. If the environment variable `LOGICALNAMEMAPPING` exists, and if the `-xl`[*d*] compiler option is set, then the compiler interprets VMS logical file names on the `INCLUDE` statement.

It sets the environment variable to a string with the following syntax.

```
"lname1=path1; lname2=path2; … "
```

Each *lname* is a logical name and each path is the path name of a directory (without a trailing "/"). All blanks are ignored when parsing this string. It strips any trailing "`/[no]list`" from the file name in the `INCLUDE` statement. Logical names in a file name are delimited by the first ":" in the VMS file name. The compiler converts file names of the form

```
lname1:file
```

to

```
path1/file
```

For logical names, uppercase/lowercase is significant. If a logical name is encountered on the INCLUDE statement which is not specified in the LOGICALNAMEMAPPING, the file name is used unchanged.

Direct I/O

Random access to files is also called direct access. A direct-access file contains a number of records that are written to or read from by referring to the record number. This record number is specified when the record is written. In a direct-access file, records must be all the same length and all the same type.

A logical record in a direct access, external file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. Unformatted, direct writes leave the unfilled part of the record undefined. Formatted, direct writes cause the unfilled record to be padded with blanks.

In using direct unformatted I/O, be careful with the number of values your program expects to read. Each READ operation acts on exactly *one* record; the number of values that the input list requires must be *less than or equal to* the number of values in that record.

Direct access READ and WRITE statements have an extra argument, REC=*n*, which gives the record number to be read or written.

Example: Direct-access, *unformatted.*

```
      OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=20,
&          FORM='UNFORMATTED', ERR=90 )
      READ( 2, REC=13, ERR=30 ) X, Y
```

This opens a file for direct-access, unformatted I/O, with a record length of 20 characters, then reads the thirteenth record as is.

Example: Direct-access, *formatted.*

```
      OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=20,
&          FORM='FORMATTED', ERR=90 )
      READ( 2, FMT="(I10,F10.3)", REC=13, ERR=30 ) A, B
```

This opens a file for direct-access, formatted I/O, with a record length of 20 characters, then reads the thirteenth record and converts it according to the format "`(I10,F10.3)`".

You can improve direct access I/O performance by opening a file with a large buffer size. You can do this with one of the options for the `OPEN` statement, the "`FILEOPT=`*fopt*" option.♦ The *fopt* itself can be "`BUFFER=`*n*." The form of the option is:

```
OPEN( …, FILEOPT="BUFFER=n", … )
```

It sets the size in bytes of the I/O buffer to use. For writes, larger buffers yield faster I/O. For good performance, make the buffer a multiple of the largest record size. This can be larger than actual physical memory, and probably the very best performance is obtained by making the record size equal to the entire file size. These larger buffer sizes may cause some extra paging. Read the *FORTRAN Reference Manual*, in the section on the `OPEN` statement.

Internal Files

An internal file is an object of type character such as a variable, substring, array, element of an array, or field of a structured record. If you are *reading* from the internal file, it can be a *constant* character string. This is called I/O, although I/O is not a precise term to use here, because you use `READ` and `WRITE` statements to deal with internal files.

- To use an internal file, give the name of the character object in place of the unit number.
- For a constant, variable, or substring, there is only a single record in the file.
- For an array, each array element is a record.
- `f77` extends direct I/O to internal files. The ANSI standard includes only sequential formatted I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings.
- Each sequential `READ` or `WRITE` starts at the beginning of an internal file.

Example: Sequential formatted read from an internal file (one record only).

```
demo$ cat intern1.f
      CHARACTER X*80
      READ( *, '(A)' ) X
      READ( X, '(I3,I4)' ) N1, N2 ! This reads the internal file X
      WRITE( *, * )  N1, N2
      END
demo$ f77 -silent intern1.f
demo$ a.out
 12 99
  12  99
demo$ ■
```

Example: Sequential formatted read from an internal file (three records).

```
demo$ cat intern3.f
      CHARACTER  LINE(4)*16
*                     12341234
      DATA  LINE(1) / ' 81  81 ' /
      DATA  LINE(2) / ' 82  82 ' /
      DATA  LINE(3) / ' 83  83 ' /
      DATA  LINE(4) / ' 84  84 ' /
      READ( LINE,'(2I4)') I,J,K,L,M,N ! This reads an internal file
      PRINT *, I, J, K, L, M, N
      END
demo$ f77  -silent intern3.f
demo$ a.out
  81  81  82  82  83  83
demo$ ■
```

Example: Direct-access read from an internal file (one record).

```
demo$ cat intern2.f
      CHARACTER LINE(4)*16
*                     12341234
      DATA LINE(1) / ' 81 81 ' /
      DATA LINE(2) / ' 82 82 ' /
      DATA LINE(3) / ' 83 83 ' /
      DATA LINE(4) / ' 84 84 ' /
      READ ( LINE, FMT=20, REC=3 ) M, N   ! This reads an internal file
20    FORMAT( I4, I4 )
      PRINT *, M, N
      END
demo$ f77  -silent intern2.f
demo$ a.out
83 83
demo$ ■
```

5.2 Tape I/O

Using tape files on UNIX systems is awkward because, historically, UNIX development was oriented toward small data sets residing on fast disks. Magnetic tape was used by early UNIX systems for archival storage and moving data between different machines. Unfortunately, many FORTRAN programs are intended to use large data sets from magnetic tape. For tape it is more reliable to use the `TOPEN()` routines than the FORTRAN I/O statements.

Using `TOPEN` for Tape I/O

A nonstandard tape I/O package (see `TOPEN` (3F)) offers a partial solution to the problem. FORTRAN programmers can transfer blocks between the tape drive and buffers declared as FORTRAN character variables. The programmer can then use internal I/O to fill and empty these buffers. This facility does not integrate with the rest of FORTRAN I/O. It even has its own set of tape logical units. For tape it is more reliable to use the `TOPEN()` routines than the FORTRAN I/O statements.

5

FORTRAN Formatted I/O for Tape

The FORTRAN I/O statements provide facilities for transparent access to *formatted*, sequential files on magnetic tape. The tape block size may be optionally controlled by the `OPEN` statement `FILEOPT` parameter. There is no bound on formatted record size, and records may span tape blocks.

FORTRAN Unformatted I/O for Tape

Using the FORTRAN I/O statements to connect a magnetic tape for *unformatted* access is less satisfactory. Note the implementation of unformatted records as a sequence of characters preceded and followed by character counts. The size of a record (+ 8 characters of overhead) cannot be bigger than the buffer size.

As long as this restriction is honored, the I/O system does not write records that span physical tape blocks, but writes short blocks when necessary. This representation of unformatted records is preserved (even though it is inappropriate for tape), so files can be freely copied between disk and tapes.

Note that since the block-spanning restriction does not apply to tape reads, files can be copied from tape to disk without any special considerations.

Tape File Representation

A FORTRAN data file is represented on tape by a sequence of data records followed by an `endfile` record. The data is grouped into blocks, the maximum size determined when the file is opened. The records are represented the same as records in disk files: formatted records are followed by newlines; unformatted records are preceded and followed by character counts. In general, there is no relation between FORTRAN records and tape blocks; that is, records can span blocks, which can contain parts of several records. The only exception is that FORTRAN won't write an unformatted record that spans blocks; thus, the size of the largest unformatted record is eight characters less than the block size.

The dd *Conversion Utility*

An endfile record in FORTRAN maps directly into a tape mark. In this respect, FORTRAN files are the same as tape system files. But since the representation of FORTRAN files on tape is the same as that used in the rest of UNIX, naive FORTRAN programs cannot read 80-column card images from tape. If you have an existing FORTRAN program and an existing data tape to read with it, translate the tape using the dd(1) utility, which adds newlines and strips trailing blanks.

Example: Convert a tape on mt0 and pipe that to the executable ftnprg.

```
demo$ dd if=/dev/rmt0 ibs=20b cbs=80 conv=unblock | ftnprg
```

The getc *Library Routine*

If you write or modify a program and don't want to use dd, you can use the getc(3F) library routine to read characters from the tape. You can then combine the characters into a character variable and use internal I/O to transfer formatted data. See also TOPEN(3F).

End-of-File

The end-of-file condition is reached when an endfile record is encountered during execution of a READ statement. The standard states that the file is positioned after the endfile record. In real life, this means that the tape read head is poised at the beginning of the next file on the tape. Thus, it would seem that you can read the next file on the tape; however, this doesn't work and is not covered by the standard.

The standard also says that a BACKSPACE or REWIND statement may be used to reposition the file. This means that after reaching end-of-file, you can backspace over the endfile record and further manipulate the file (such as writing more records at the end), rewind the file, and reread or rewrite it.

Access on Multiple-file Tapes

Each tape drive can be opened by many names. The name used determines certain characteristics of the connection, which are the recording density and whether the tape is automatically rewound when opened and closed.

To access a file on a multiple-file tape, use the `mt`(1) utility to position the tape to the correct file, then open the file as a no-rewind magnetic tape such as `/dev/nrmt0`. Using the tape with this name also prevents it from being repositioned when it is closed. If your program reads the file until end-of-file, then reopens it, it can access the next file on the tape. Any following programs can access the tape where you left it (preferably at the beginning of a file, or past the endfile record). If your program terminates prematurely it can leave the tape positioned in an unpredictable place.

Program Development 6

This chapter is organized into the following sections.

Simple Program Builds	*page 99*
Program Builds with the make Program	*page 100*
Tracking and Controlling Changes with SCCS	*page 105*

6.1 Simple Program Builds

For a program that depends on only a single source file and some system libraries, you can compile all of the source files every time you change the program. Even in this simple case, the `f77` command can involve much typing, and with options or libraries, a lot to remember. A script or alias can help.

Writing a Script

You can write a shell script to save typing. For example, to compile a small program that is in the file `example.f`, and that uses the SunCore® graphics library, you can save a one-line shell script onto a file called `fex`, that looks like this.

```
f77 example.f –lcore77 –lcore –o example
```

You may need to put execution permissions on `fex`.

```
demo$ chmod +x fex
```

Creating an Alias

You can create an alias to do the same command.

```
demo$ alias fex "f77 example.f –lcore77 \
–lcore –o example"
```

Using a Script or Alias

Either way, to recompile `example.f`, you type only `fex`.

```
demo$ fex
```

Limitations

With multiple source files, forgetting one compile makes the objects inconsistent with the source. Recompiling all files after every editing session wastes time, since not every source file needs recompiling. Forgetting an option or a library produces questionable executables.

6.2 Program Builds with the `make` *Program*

The `make` program recompiles only what needs recompiling, and it uses only the options and libraries you want. This section shows you how to use normal, basic `make`, and it provides a simple example. For a summary, see `make` (1).

The `makefile`

The way you tell `make` what files depend on other files, and what processes to apply to which files, is to put this information into a file called the `makefile`, in the directory where you are developing the program.

Example: Suppose you have a program of four source files and a `makefile`.

```
demo$ ls
makefile
commonblock
computepts.f
pattern.f
startupcore.f
demo$ ■
```

Assume both `pattern.f` and `computepts.f` do an `include` of `commonblock`, and you wish to compile each `.f` file and link the three relocatable files (plus a series of libraries) into a program called `pattern`.

The `makefile` for this example is listed below.

```
demo$ cat makefile
pattern: pattern.o computepts.o startupcore.o
     f77 pattern.o computepts.o startupcore.o –lcore77 \
     –lcore –lsunwindow –lpixrect –o pattern
pattern.o: pattern.f commonblock
     f77 –c –u pattern.f
computepts.o: computepts.f commonblock
     f77 –c –u computepts.f
startupcore.o: startupcore.f
     f77 –c –u startupcore.f
demo$ ■
```

The first line of this `makefile` says:

- `make pattern`
- `pattern` depends on `pattern.o`, `computepts.o`, and `startupcore.o`

The second line is the command for making `pattern`.

The third line is a continuation of the second.

There are four such paragraphs or entries in this `makefile`. The structure of these entries is:

- *Dependencies* — Each entry starts with a line that names the file to `make`, and names all the files it depends on.
- *Commands* — Each entry has one or more subsequent lines that contain Bourne shell commands, and that tell how to build the target file for this entry. These subsequent lines must each be indented by a tab.

6

Using `make`

The `make` command can be invoked with no arguments, such as this.

```
demo$ make
```

The `make` utility looks for a file named `makefile` or `Makefile` in the current directory, and takes its instructions from there.

The `make` utility general actions are:

- From the `makefile`, it gets all the target files it must `make`, and what files they depend on. It also gets the commands used to `make` the target files.
- It gets the date and time changed for each file.
- If any target file is not up to date with the files it depends on, then that target is rebuilt, using the commands from the `makefile` for that target.

The C Preprocessor

You can use the C preprocessor for such things as passing strings to `f77`.

Example: If you want your program to print the time it was compiled when it is given a command-line argument of `-v`, then you need to add code that looks like this.

```
      IF (ARGSTRING .EQ. "-v") THEN
           PRINT *, CTIME
           CALL EXIT(0)
      END IF
```

This example is just an extension of the `make` example with pattern.f.

Use the C preprocessor to define `CTIME` as a quoted string that can be printed. The next two examples show how to do this.

The C preprocessor is applied if the file names have the suffix `.F`, so we change the file name.

```
demo$ mv pattern.f pattern.F
```

The –D option defines a name to have a specified value for the C preprocessor, as if by a #define line. So we change the compilation line for pattern.F in the makefile as follows (which works only in sh).

```
demo$ f77 "–DCTIME=\"‘date‘\"" –c –u pattern.F
```

The part up to the -c option gets the output of the date command, puts quotes around it, stuffs that into CTIME, and passes that on to the C preprocessor. If you do not want the gory details, skip the next paragraph.

The innermost single quotes are backquotes or grave accents. They indicate that the output of the command contained in them (in this case the date command) is to be substituted in place of the backquoted word(s). The next level of quote marks is what makes this define a FORTRAN quoted string, so it can be used in the print statement. These marks must be escaped (or quoted) by preceding backslashes because they are nested inside another pair of quote marks. The outermost marks indicate to the interpreting shell that the enclosed characters are to be interpreted as a single argument to the f77 command. They are necessary because the output of the date command contains blanks, so that without the outermost quoting it would be interpreted as several arguments, which would not be acceptable to f77.

The preprocessor now converts CTIME to "jan15…", so that

```
      PRINT *, CTIME
```

becomes

```
      PRINT *, "jan15…"
```

The purpose here is to show *how* such strings are passed to the C preprocessor; the particular string passed is not useful, but the *method* is the same.

Macros with make

The make program does simple parameterless *macro* substitutions. In the make example above, the list of relocatable files that go into the target program pattern appears twice: once in the dependencies and once in the f77 command that follows. This makes changing the makefile error-prone, since the same changes must be made in two places in the file. In this case, you can add the following to the beginning of your makefile.

Sample macro definition.

```
OBJ = pattern.o computepts.o startupcore.o
```

Change the description of the program `pattern` into:

Sample macro use.

```
pattern: $(OBJ)
     f77 $(OBJ) –lcore77 –lcore –lsunwindow \
     –lpixrect –o pattern
```

Note the peculiar syntax in the above example: a *use* of a macro is indicated by a dollar sign immediately followed by the name of the macro in parentheses. For macros with single-letter names, the parentheses may be omitted.

Overriding Macro Values

The initial values of `make` macros can be overridden with command-line options to `make`. Add the following line to the top of the `makefile`.

```
FFLAGS=–u
```

Change each command for making FORTRAN source files into relocatable files by deleting that flag, the compilation of `computepts.f` looks like this.

```
f77 $(FFLAGS) –c computepts.f
```

The final link looks like this.

```
f77 $(FFLAGS) $(OBJ) –lcore77 –lcore –lsunwindow \
     lpixrect –o pattern
```

If you issue the bare `make` command, everything compiles as before. However, the following command does more.

```
demo$ make "FFLAGS=–u –O"
```

Above, the `-O` flag, as well as the `-u` flag, is passed to `f77`.

Suffix Rules in `make`

If you don't tell `make` how to `make` a relocatable file, it uses one of its default `rules`, in this case:

> Use the `f77` compiler; pass as arguments any flags specified by the `FFLAGS` macro, the `-c` flag, and the name of the source file to be compiled.

You can take advantage of this rule twice in the example, but still must explicitly state the dependencies, and the nonstandard command for compiling the `pattern.f` file. The `makefile` is as follows.

```
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=–u
pattern: $(OBJ)
     f77 $(OBJ) –lcore77 –lcore –lsunwindow \
     –lpixrect –o pattern
pattern.o: pattern.f commonblock
     f77 $(FFLAGS) "–DCTIME=\"‘date‘\"" –c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

6.3 Tracking and Controlling Changes with SCCS

SCCS is Source Code Control System. It provides a way to:

- Keep track of the evolution of a source file (change history)
- Prevent different programmers from changing the same source file at the same time
- Keep track of the version number by providing version stamps

The basic three operations of SCCS are putting files under SCCS control, checking out a file for editing, and checking in a file. This section shows you how to use SCCS to do these things and provides a simple example, using the previous program. It describes normal, basic SCCS, and introduces only three SCCS commands: `create`, `edit`, and `delget`.

6

Putting Files under SCCS

Putting files under SCCS control involves making the SCCS directory, inserting SCCS ID keywords into the files (optional), and creating the SCCS files.

Making the SCCS Directory

To begin, you must create the SCCS subdirectory in the directory in which your program is being developed.

```
demo$ mkdir SCCS
demo$ ■
```

The ‘SCCS’ must be uppercase.

Inserting SCCS ID Keywords

Some people put one or more SCCS ID keywords into each file, but that is optional. These will later be filled in with a version number each time the file is checked in with a `get` or `delget` SCCS command. There are three likely places to put such strings:

- Comment lines
- Parameter statements
- Initialized data

The advantage of the last is that the version information appears in the compiled object program, and can be printed using the `what` command. Included header files containing only parameter and data definition statements do not generate any initialized data, so the keywords for those files usually are put in comments or in parameter statements. Finally, in the case of some files, like ASCII data files or makefiles, the source is all there is, so the SCCS information can go in comments, if anywhere.

Identify the makefile with a `make` comment containing the keywords.

```
#     %Z%%M% %I% %E%
```

The source files `startupcore.f` and `computepts.f` and `pattern.f` can be identified by initialized data of the form.

```
      CHARACTER*50 SCCSID
      DATA SCCSID/"%Z%%M% %I% %E%\n"/
```

You can also replace the word `CTIME` by a parameter that is automatically updated whenever the file is accessed with `get`.

```
      CHARACTER*(*) CTIME
      PARAMETER ( CTIME="%E%")
```

Remove the `-DCTIME` from the `makefile`. Finally, the included file `commonblock` is annotated with a FORTRAN comment.

```
C     %Z%%M% %I% %E%
```

Creating SCCS *Files*

Now you can put these files under control of SCCS with the SCCS `create` command.

```
demo$ sccs create makefile commonblock startupcore.f \
          computepts.f pattern.f
demo$ ■
```

Your files now look like this after SCCS keyword expansion.
`makefile`

```
#    @(#)makefile1.184/03/01
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
     f77 $(OBJ) –lcore77 –lcore –lsunwindow \
          –lpixrect –o pattern
pattern.o: pattern.f commonblock
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

commonblock

```
C     @(#)commonblock1.184/03/01
      INTEGER NMAX, NPOINTS
      REAL X, Y
      PARAMETER ( NMAX = 200 )
      COMMON NPOINTS
      COMMON X(NMAX), Y(NMAX)
```

computepts.f

```
      SUBROUTINE COMPUTEPTS
      DOUBLE PRECISION T, DT, PI
      PARAMETER ( PI=3.1415927 )
      INCLUDE 'commonblock'
      INTEGER I
      CHARACTER*50 SCCSID
      DATA SCCSID/"@(#)computepts.f1.184/03/05\n"/
c     Compute x/y coordinates of NPOINTS points
c     on a unit circle as index I moves from 1 to
c     NPOINTS, parameter T sweeps from 0 to
c     PI(2 + NPOINTS/2) in increments of
c     (PI/2)*(1 + 4/NPOINTS)
      T = 0.0
      DT = (PI/2.0)*(1.0 + 4.0/DBLE(NPOINTS))
      DO 10 I = 1, NPOINTS+1
           X(I) = COS(T)
           Y(I) = SIN(T)
           T = T+DT
10    CONTINUE
      RETURN
      END
```

startupcore.f

```
      SUBROUTINE STARTUPCORE
      INCLUDE '/usr/include/f77/usercore77.h'
C      Make initializing calls to core library
      COMMON /VWSURF/ VSURF
      INTEGER VSURF(VWSURFSIZE), SELECTVWSURF
      INTEGER PIXWINDD, INITIALIZECORE, INITIALIZEVWSURF
C      (Use CGPIXWINDD instead of PIXWINDD for color)
      EXTERNAL PIXWINDD
      CHARACTER*4 ENVRETURN
      CHARACTER*50 SCCSID
      INTEGER LOC
      DATA SCCSID/"@(#)startupcore.f 1.1 84/03/05\n"/
      DATA VSURF /VWSURFSIZE*0/

      VSURF(DDINDEX) = LOC(PIXWINDD)
      IF (INITIALIZECORE(BASIC, NOINPUT, TWOD) .NE. 0)
     &      CALL EXIT
      CALL GETENV( "window_me", ENVRETURN )
      IF (ENVRETURN .EQ. " ") THEN
      WRITE(0,*) "Must run in a window"
      CALL EXIT(2)
      ENDIF
      IF (INITIALIZEVWSURF( VSURF, FALSE) .NE. 0)
     &      CALL EXIT(2)
      IF (SELECTVWSURF(VSURF) .NE. 0) CALL EXIT(3)
      CALL SETWINDOW( -1.5, 1.5, -2.0, 2.0 )
      CALL CREATETEMPSEG()
      RETURN
      END

      SUBROUTINE CLOSECORE
      INCLUDE '/usr/include/f77/usercore77.h'
C     Make terminating calls to core library
      COMMON /VWSURF/ VSURF
      INTEGER VSURF(VWSURFSIZE)

      CALL CLOSETEMPSEG()
      CALL DESELECTVWSURF( VSURF )
      CALL TERMINATECORE()
      RETURN
      END
```

pattern.f

```
      PROGRAM STAR
C     Draw a star of n points, arg n
      INCLUDE 'COMMONBLOCK'
      CHARACTER*10 ARG
      INTEGER I, IARGC, LNBLNK
      CHARACTER*(*) CTIME
      PARAMETER ( CTIME="84/03/05" )
      CHARACTER*50 SCCSID
      DATA SCCSID/"@(#)pattern.f1.184/03/05\n"/

      IF (IARGC() .LT. 1 ) THEN
           CALL GETARG( 0, ARG)
           I = LNBLNK(ARG)
           WRITE (0,*) "Usage: ",arg(:i)," -v or ",arg(:i)," nnn"
           CALL EXIT (0)
      END IF
      CALL GETARG( 1, ARG )
      IF (ARG .EQ. "-v") THEN
           PRINT *, CTIME
           CALL EXIT(0)
      END IF
      READ( ARG, '(I3)') NPOINTS
      NPOINTS = NPOINTS*4
      IF (NPOINTS .LE. 0 .OR. NPOINTS .GT. NMAX-1) THEN
         WRITE(0,*) NPOINTS/4, "Out of range [1..",(NMAX-1)/4,"]"
         CALL EXIT(12)
      END IF
      CALL COMPUTEPTS
      CALL STARTUPCORE
      CALL MOVEABS2( X(1),Y(1) )
      CALL POLYLINEABS2( X(2), Y(2), NPOINTS)
      PAUSE
      CALL CLOSECORE
      END
```

Of course, this is an example of how SCCS operates rather than how it is really used. You don't need the preprocessor any longer to drop in the compilation date. The `-v` argument is without purpose, since you can use the `what` command, which gives you much more detail.

Checking Files Out and In

Out— Once your source code is under SCCS control, you use SCCS for two main tasks: to *check out* a file so that you can edit it and to *check in* a file you are done editing. A file is checked out using the SCCS `edit` command.

Example: Check out a file using SCCS.

```
demo$ sccs edit computepts.f
demo$
```

In this example, SCCS makes a writable copy of `computepts.f` in the current directory, and records your login name. Other users cannot check it out while you have it checked out, but they *can* find who has checked out which files.

In— Check in the file with the `sccs delget` command when you have completed your current editing.

Example: Check in a file using SCCS.

```
demo$ sccs delget computepts.f
demo$
```

This causes the SCCS system to do the following:

1. Make sure that you are the user who checked it out (compares login names).
2. Solicit a descriptive comment from the user on the changes.
3. Make a record of what was changed in this editing session.
4. Delete the writable copy of `computepts.f` from the current directory.
5. Replace it by a read-only copy with the SCCS keywords expanded.

The SCCS command `delget` is a composite of the two simpler SCCS commands, `delta` and `get`. The `delta` command does the first three items in the list above and the `get` command does the fourth and fifth.

6

Creating and Using Libraries 7

This chapter is organized into the following sections.

Libraries in General	*page 113*
Library Search Paths and Order	*page 117*
Static Libraries	*page 122*
Dynamic Libraries	*page 125*
Libraries Provided	*page 135*

7.1 Libraries in General

A software *library* is usually a set of subprograms. Each member of the set is called a library *element* or *module.* A *relocatable* library is one whose elements are relocatable (`.o`) files. These object modules are inserted into the executable file by the linker during the compile/link sequence. See `ld`(1).

There are two basic kinds of software libraries—static and dynamic.

- S*tatic* library: modules are bound into the executable file *before* execution.
- *Dynamic* library: modules can be bound in *after* execution begins.

Some examples of *static* libraries on the system are:

- FORTRAN library: `libF77.a`
- VMS FORTRAN library: `libV77.a`
- Math library: `libm.a`
- C library: `libc.a`

Some examples of *dynamic* libraries on the system are:

- FORTRAN library: `libF77.so`
- VMS FORTRAN library: `libV77.so`
- C library: `libc.so`

Advantages of Libraries

Relocatable libraries provide an easy way for commonly used subroutines to be used by several programs. The programmer need only name the library when linking the program, and those library modules that resolve references in the program are linked—copied into the executable file. This has two advantages.

- Only the needed modules are loaded.
- The programmer need not change the link command line as subroutine calls are added and removed during program development.

Disadvantages of Libraries

When the linker *searches* a library, it extracts elements whose entry points are referenced in other parts of the program it is linking, such as subprogram or entry names or names of `COMMON` blocks initialized in `BLOCKDATA` subprograms.

- *The whole thing* — When the linker extracts a library element, it takes the whole thing. Since an element corresponds to the result of a compilation, this means that routines that are compiled together are always linked together. This is a difference between this operating system and some other systems and may affect the way you divide up your libraries.
- *Order matters* — One difference between this operating system and others is that in linking, order really matters. The linker processes its input files in the order that they appear on the command line—left to right. When the linker decides whether or not a library element is to be linked, its decision is based only on the relocatable modules it has already processed.

  Example: *Order matters*–If the FORTRAN program is in two files, `main.f` and `graf.f`, and only the latter accesses the SunCore graphics library, it is an error to reference that library before `graf.f` or `graf.o`:

```
(Wrong) demo$f77 main.f -lcore77 -lcore graf.f -o myprog
(Right) demo$f77 main.f graf.f -lcore77 -lcore -o myprog
```

Debug Aids

You can ask the linker various questions about libraries—how they are being used, what paths are being searched for libraries, and so forth.

Load Map

To display a load map, pass the load map option to the linker by `-Qoption`. This displays which libraries are linked and which routines are obtained from which libraries during the creation of the executable module.

Example: `-m` for load map.

Solaris 2.x

```
demo$ f77 -Qoption ld -m any.f77
```

Example: `-M` for load map.

Solaris 1.x

```
demo$ f77 -Qoption ld -M any.f77
```

Other Queries

For Solaris 2.3 and later, there are linker debugging aids which help diagnose some linking problems. One way to get the list is "`-Qoption ld -Dhelp`"

Example: List some linker debugging aid options.

Solaris 2.3

```
demo$ f77 -Qoption ld -Dhelp any.f
      …
debug: files    display input file processing (files and libraries)
debug: help     display this help message
debug: libs     display library search paths; detail flag shows actual
debug:          library lookup (-l) processing
      …
demo$
```

Read "***Linker and Libraries Manual***" for details.

Consistent Compile and Link

Do not build libraries with inconsistent modules. Some options require consistent compiling and linking. Inconsistent compilation and linkage is not supported. See "Consistent Compile and Link," on page 23, for the options and steps involved.

Fast Directory Cache for the Link-editor

Solaris 1.x

For Solaris 1.x only, the `ldconfig` utility configures a performance-enhancing cache for the `ld.so` runtime link-editor. It is run automatically from the `/etc/rc.local` file. For best performance, you should run it manually after you install a new shared object (for example, a shared library), and every time the system is rebooted thereafter.

If you don't want to run it manually at each reboot the system, then add the name of the shared libraries directory to the `ldconfig` line near the end of the `rc.local` file; do this on the machine where your compiler is installed, and on any client machines. Then you must run it manually once on each client.

Set the `ldconfig` path differently for standard / nonstandard installations:

- If you installed in the *standard* location, put that location in `rc.local`.

  Example: Configure performance-enhancing cache—*standard* install location.

Using the `vi` editor, it should look something like this.

Add the `/usr/lang/SC3.0.1/` directory to the `ldconfig` line near the end of the `rc.local` file.

The "`3.0.1`" in `SC3.0.1` varies with the release number, of course.

```
demo% su root
Password: root-password
demo# vi /etc/rc.local
          ...
# Build the link-editor fast directory cache.
#
if [ -f /usr/etc/ldconfig ]; then
    ldconfig  /usr/lang/SC3.0.1; (echo "cache") > /dev/console
fi
:wq
demo#
```

- If you installed into the nonstandard */your/dir/* location, use that path.

  Example: Configure performance-enhancing cache—*nonstandard* install.

```
...
     ldconfig  /your/dir/SC3.0.1
...
```

7.2 Library Search Paths and Order

The linker searches for libraries in several locations, and it searches in certain prescribed orders. Some of these locations are standard locations, some depend on the options `-l`*x* and `-L`*dir*, and some depend on the environment variasbles `LD_RUN_PATH` or `LD_LIBRARY_PATH`. You can make some changes to the order and locations.

Order of Paths Critical for Compile (Solaris 1.x)

In Solaris 1.x, if you specify library search paths, the *order* of the paths can be critical. The compilation can fail if you cause an incompatible version of the math library, `libm`, to be used.

Symptom

Some entry is missing. The error message looks like the following.

Solaris 1.x

```
demo$ f77 test.f
test.f:
 MAIN:
ld: Undefined symbol
   _ _ _start_libm
  <other entries>
demo$ ■
```

Correct Order

To get a *compatible* version of `libm` do the following.

If `/usr/lib` is in `LD_LIBRARY_PATH`, then:

- If installation was to `/usr/lang/` (*standard* installation), then put `/usr/lang/lib` in `LD_LIBRARY_PATH` *before* `/usr/lib`
- If installation was to `/`*my*`/`*dir*`/` (*nonstandard* installation), then put `/`*my*`/`*dir*`/lib` in `LD_LIBRARY_PATH` *before* `/usr/lib`

Otherwise an incompatible version of the math library, `libm`, is used.

Usage of `LD_LIBRARY_PATH` is not generally recommended, but sometimes there may be no other way.

Do Not Use `-L/usr/lib`

In Solaris 1.x, do not use the `-L`*dir* option to specify `/usr/lib`, because then you get an incompatible version of the math library, `libm`. You never need to use the `-L`*dir* option to specify `/usr/lib`, because you always get `/usr/lib` by default.

Library not Found

In some circumstances the dynamic linker cannot find some libraries.

Symptom

The following error message is displayed during program execution.

```
ld.so: library not found
```

This happens during the running of `a.out`, not during compilation or linking.

Some Causes

- Creating an executable using dynamic libraries, and moving the libraries

  One instance: Build `a.out` with your own dynamic libraries in */my/libs/*, then you move the libraries.

- Replacing all paths in `LD_LIBRARY_PATH` with one directory

  One instance: Use OpenWindows and define the `LD_LIBRARY_PATH` environment variable to link in the Xview libraries only.

Prevention

Set `LD_LIBRARY_PATH` to *include* the path where the missing library resides, instead of setting it to be only the one path.

Example: Put */my/libs/* into LD_LIBRARY_PATH in front of what is there.

sh

```
demo$ LD_LIBRARY_PATH=/my/libs/:$LD_LIBRARY_PATH
demo$ export LD_LIBRARY_PATH
```

csh

```
demo% setenv LD_LIBRARY_PATH /my/libs/:$LD_LIBRARY_PATH
```

Order on the Command Line for –lx Options

For any particular unresolved reference, libraries are searched only once, and only for symbols that are undefined at that point in the search. If you list more than one library on the command line, then the libraries are searched in the order they are found on the command line. So the order on the command line makes a difference.

Therefore, place `-lx` options as follows:

- Place the `–lx` option after any `.f`, `.for`, `.F`, or `.o` files.
- If you call functions in `libx`, and they reference functions in `liby`, then place `-lx` before `-ly`.

Search Order for Library Search Paths

Linker library search paths can depend on the following:

- Solaris 1.x/2.x
- Installation: standard location or */my/dir/*
- Building or running the executable file

The base directory, here called *BaseDir*, is defined as follows:

Table 7-1 BaseDir for Library Search Paths

BaseDir Defined

	Standard Install	**Nonstandard Install to** */my/dir/*
Solaris 1.x	`/usr/lang/`	*/my/dir/*
Solaris 2.x	`/opt/SUNWspro/`	*/my/dir*`/SUNWspro/`

Building the executable file

While *building* the executable file, the *static* linker searches for *any* libraries in the following paths (among others), in the specified order.

Search paths in terms of ***BaseDir***

Solaris 1.x	*/BaseDir*`/lib/`	Sun shared libraries here
	/BaseDir`/SC3.0.1/lib/`	Sun libraries, shared or static, here
	`/usr/lang/lib/`	Standard location for Sun software
	`/usr/lib/`	Standard location for UNIX software
Solaris 2.x	*/BaseDir*`/lib/`	Sun shared libraries here
	/BaseDir`/SC3.0.1/lib/`	Sun libraries, shared or static, here
	`/opt/SUNWspro/lib/`	Standard location for Sun software
	`/usr/ccs/lib/`	Standard location for SVr4 software
	`/usr/lib`	Standard location for UNIX software

For both Solaris 1.x and 2.x, these directories are the ones searched without the user specifying them; they are generally what is meant by the ***default directories***

While *building* the executable file (1.x and 2.x):

- The static linker searches paths specified by `LD_LIBRARY_PATH`. For the search order relative to the above paths, see `ld`(1).
- The static linker searches paths specified by `-L`*dir*. For the search order relative to `LD_LIBRARY_PATH`, see `ld`(1).
- In general, it is best to avoid using `LD_LIBRARY_PATH` if at all possible.

Running the executable file

While *running* the executable file, the *dynamic* linker searches for *shared* libraries in these paths (among others), in the specified order.

Search paths in terms of ***BaseDir***

For both Solaris 1.x and 2.x, these directories are the ones searched without the user specifying them; they are generally what is meant by the ***default directories***

Solaris 1.x	*/BaseDir*/lib/	Sun shared libraries here
	/BaseDir/SC3.0.1/lib/	Sun libraries, shared or static, here
	/usr/lang/lib/	Standard location for Sun software
	/usr/lib/	Standard location for UNIX software
Solaris 2.x	*/BaseDir*/lib/	Built in by driver, unless -norunpath
	/opt/SUNWspro/lib	Built in by driver, unless -norunpath
	Other paths built in by -R or LD_RUN_PATH when the executable was generated	Uses paths stored in the executable. Ignores current (runtime) values of -R and LD_RUN_PATH.
	/usr/lib/	Standard location for UNIX software

- LD_LIBRARY_PATH (*Solaris 1.x and 2.x*)

  While *running* the executable file:
  - The *dynamic* linker searches paths specified by LD_LIBRARY_PATH. For the search order relative to the above paths, see ld(1).
  - LD_LIBRARY_PATH can change after the executable was created. No matter what the value of LD_LIBRARY_PATH was while the executable file was being built, the value at runtime is used while the executable is running. To see which directories were built in when the executable was created, use dump, as shown below.

    Example: In *Solaris 2.x*, list the directories embedded in a.out.

```
demo$ dump -Lv a.out | grep RPATH   {No comparable utility for 1.x}
```

  - In general, it is best to avoid using LD_LIBRARY_PATH if at all possible.

- LD_RUN_PATH or -R: Paths Stored in Executable are Used (*Solaris 2.x*)

  While *running* the executable file:
  - The *dynamic* linker searches paths that had been specified by LD_RUN_PATH or -R while the executable file was being generated.
  - The current values of LD_RUN_PATH and -R are ignored. For f77, -R and LD_RUN_PATH are not identical; see *–R list*, page 54, for differences.

7.3 Static Libraries

Static libraries are built from object files (.o files) using the program `ar`.

If you build a *static* library, then elements in the library must not reference entry points defined in elements that precede them, since these libraries are searched in presentation order (that is, front to back). You can use `lorder` and `tsort` to order static libraries.

Sample Creation of a Static Library

Example: Create a static library from four subroutines in one file.

Routines for library

```
demo$ cat one.f
      subroutine twice ( a, r )
      real a, r
      r = a * 2.0
      return
      end
      subroutine half ( a, r )
      real a, r
      r = a / 2.0
      return
      end
      subroutine thrice ( a, r )
      real a, r
      r = a * 3.0
      return
      end
      subroutine third ( a, r )
      real a, r
      r = a / 3.0
      return
      end
demo$ ■
```

Example: This main program uses one of the subroutines in the library.

Main

```
demo$ cat teslib.f
      read(*,*) x
      call twice( x, z )
      write(*,*) z
      end
demo$ ■
```

Sample Creation of a Static Library (continued)

Split the file, using `fsplit`, so there is one subroutine per file.

```
demo$ fsplit one.f
twice.f
half.f
thrice.f
third.f
demo$ ■
```

Compile each with the `-c` option so it will *compile only*, and leave the `.o` files.

```
demo$ f77 -c half.f
half.f:
half:
demo$ f77 -c third.f
third.f:
third:
demo$ f77 -c thrice.f
thrice.f:
thrice
demo$ f77 -c twice.f
twice.f:
twice:
demo$ ■
```

Create a static library using `ar`.

```
demo$ ar cr faclib.a half.o third.o thrice.o twice.o
```

Above, tell `ar` to create static library `faclib.a` from the four object files.

Alternate: You can specify any order using `lorder` and `tsort`.

```
demo$ ar cr faclib.a 'lorder half.third.o thrice.o twice.o | tsort'
```

In Solaris 1.x, use `ranlib` to randomize the static library.

Solaris 1.x

```
demo$ ranlib faclib.a
```

Do not do this in Solaris 2.x

Sample Creation of a Static Library (continued)

To use this new library, put the file name in the compile command. No special flag is needed—the linker recognizes a library when it encounters one.

Example: Use the new library while compiling the main program.

Put file name in compile command.

```
demo$ f77 teslib.f faclib.a
teslib.f:
MAIN:
demo$ ■
```

Example: Check names—use `nm` to list names of all objects in executable file.

This output format is for Solaris 2.x. It may vary for other releases.

`twice` appears→ `half`, `third`, `thrice` do not appear.

grep confirms that `twice` appears and `half`, `third`, `thrice` do not appear. →

```
demo$ nm a.out
[Index]   Value      Size    Type  Bind  Other Shndx   Name
[1]  |         0|       0|FILE |LOCL |0    |ABS    |a.out
…                          ← many lines not shown
[25] |         0|       0|FILE |LOCL |0    |ABS    |crti.s
[26] |         0|       0|FILE |LOCL |0    |ABS    |crt1.s
[27] |    189048|       0|NOTY |LOCL |0    |13     |v.16
[28] |    189024|       0|NOTY |LOCL |0    |13     |v.17
…                          ← many lines not shown
[190]|    193950|       1|OBJT |GLOB |0    |17     |__cblank
[191]|     77668|     164|FUNC |GLOB |0    |8      |MAIN_
…                          ← many lines not shown
[260]|     77832|      72|FUNC |GLOB |0    |8      |twice_
[261]|    194088|       4|OBJT |GLOB |0    |17     |_fp_current_exceptions
[262]|    188904|       0|FUNC |GLOB |0    |UNDEF  |close
[263]|    106400|      40|FUNC |GLOB |0    |8      |__rungetc
[264]|    113432|     784|FUNC |GLOB |0    |8      |__prnt_ext
[266]|    119624|     928|FUNC |WEAK |0    |8      |ieee_handler
…                          ← many lines not shown
demo$ nm a.out | grep twice
[260]|     77832|      72|FUNC |GLOB |0    |8      |twice_
demo$ nm a.out | grep half
demo$ nm a.out | grep third
demo$ nm a.out | grep thrice
demo$ ■
```

Example: Test the executable file—run `a.out`

```
demo$ a.out
6
    12.0000
demo$ ■
```

Sample Replacement in a Static Library

If you recompile an element of a static library (usually because you changed the source), replace it in its library by running `ar` again.

Example: Recompile, replace. Give `ar` the `r` option (use `cr` only for creating).

```
demo$ f77 -c half.f
demo$ ar r faclib.a half.o
demo$ ■
```

7.4 Dynamic Libraries

The defining aspect of a *dynamic* library is that modules can be bound into the executable file *after* execution begins.

Perhaps its most useful feature is that a module can be used by various executing programs *without* duplicating that module in each and every one of them. For this reason a dynamic library is also called a *shared* library, or sometimes a *dynamic shared* library.

Dynamic Library Features

- A dynamic library is a set of object modules, each in executable file format (the `a.out` format), but the set has no main entry.
- The object modules are *not* bound into the executable file by the linker during the compile/link sequence; such binding is deferred until runtime.
- A shared library module is bound once into the first running program that references it. If any subsequent running program references it, that reference is mapped to this first copy.
- If you change a module of a shared library, then whenever any application using it starts to execute, it will get the changed version.
  - Advantage: Ease of maintaining programs
  - Disadvantage: Users can get different results from *unchanged* executable (or from what appears to them as an unchanged executable)

Performance Issues for Dynamic Library

There is the usual trade-off between space and time.

- Less space

  Deferring the binding of the library module:
  - In general saves some *disk* space
  - Can reduce demand on *processor memory* when several processes using the library are active simultaneously.

- More time

  It takes a little more CPU time to do the following:
  - Load the library during runtime.
  - Do the link editing operations.
  - Execute the library position-independent code.

- Possible time savings
  - If the library module your program needs is already loaded and mapped because another running program referenced it, then the extra CPU time used can be offset by the savings in I/O access time. If the extra CPU time is less than or equal to the saved I/O time, then you get performance that is the same or better.
  - You can get more bang for the buck in an environment where multiple processes using the library are active simultaneously, that is, when the library is actually being shared. The extra bang comes from a reduction in working set size.

- Overall Speedup? Programs Vary

  Because of these various performance issues, some programs are faster with *shared* libraries, some are faster with *nonshared* libraries. You can bind each way to see if one way is significantly better *for your program*.

Position-Independent Code and `-pic`

Position-independent code (PIC) is code that can be bound to any address in a program without requiring relocation by the link editor. Since the code does not need the customizations created by such relocation, the code is inherently sharable between multiple processors. Thus, if you are building code to be part of a shared library, you must make it position-independent code.

The `-pic` compiler option produces position-independent code. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8K on SPARC processors. The `-PIC` compiler option is similar to `-pic`, but allows the global offset table to span the range of 32-bit addresses.

Binding Options

You can specify the binding option when you compile, that is, *dynamic* or *static* libraries. These options are actually linker options, but they are recognized by the compiler and passed on to the linker.

See “–Bbinding,” on page 32, and “–dbinding” page 34.

If you provide a library for your customers, then providing both a dynamic and a static version allows the customers the flexibility of binding whichever way is best for their application. For example, if the customer is doing some benchmarks, the `–dn` option reduces one element of variability.

7

A Simple Dynamic Library

If you compile the source files with `-pic` or `-PIC`, then you can build a dynamic library from the relocatable object (`.o`) files using the `ld` command.

Sample Create of a Dynamic Library (2.x)

Solaris 2.x

Start with the same files used for the static library example: `half.f`, `third.f`, `thrice.f`, `twice.f`.

Example: Compile with `-pic`.

```
demo$ f77 -pic -c -silent *.f
demo$ ■
```

Example: Link, and specify the `.so` file, and the `-h` to get a version number.

```
demo$ ld -o libfac.so.1 -dy -G -h libfac.so.1 -z text *.o
demo$ ■
```

The `-G` tells the linker to build a dynamic library.

The "`-z text`" warns you if it finds anything other than position-independent code, such as relocatable text. It does not warn you if it finds writable data.

Example: Bind: make the executable file `a.out`.

```
demo$ f77 teslib.f libfac.so.1
teslib.f:
 MAIN:
demo$ ■
```

Example: Run.

```
demo$ a.out
6
     12.0000
demo$ ■
```

Sample Create of Dynamic Library in 2.x (continued)

Inspect the `a.out` file for use of shared libraries. The `file` command shows that `a.out` is a dynamically linked executable — programs that use shared libraries are completely link-edited during execution.

Example: Use the `file` command to see if `a.out` is dynamically linked.

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo$ file a.out
a.out: ELF 32-bit MSB executable SPARC Version 1
dynamically linked, not stripped
demo$ ■
```

The `ldd` command shows that `a.out` uses some shared libraries, including `libfac.so.1` and `libc` (included by default by `f77`). It also shows exactly which files on the system will be used for these libraries.

Example: Use the `ldd` command to see if `a.out` uses shared libraries.

```
demo$ ldd a.out
        libfac.so.1 =>   ./libfac.so.1
        libF77.so.2 =>   /opt/SUNWspro/lib/libF77.so.2
        libc.so.1 =>     /usr/lib/libc.so.1
        libucb.so.1 =>   /usr/ucblib/libucb.so.1
        libresolv.so.1 =>        /usr/lib/libresolv.so.1
        libsocket.so.1 =>        /usr/lib/libsocket.so.1
        libnsl.so.1 =>   /usr/lib/libnsl.so.1
        libelf.so.1 =>   /usr/lib/libelf.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libintl.so.1 =>  /usr/lib/libintl.so.1
        libw.so.1 =>     /usr/lib/libw.so.1
demo$ ■
```

Your path may vary.

Sample Create of a Dynamic Library (1.x)

Solaris 1.x

Start with the same files used for the static library example: `half.f`, `third.f`, `thrice.f`, `twice.f`. This library is very simple in that it consists of procedures *only*— *no* global data is exported (made available for direct reference by programs using the library).

Example: Compile with `-pic`.

```
demo$ f77 -silent -pic -c half.f third.f thrice.f twice.f
demo$ ■
```

Example: Link, and specify the `.so` file and version number.

```
demo$ ld -o libfac.so.1.1 -Bdynamic -assert pure-text *.o
demo$ ■
```

The "`-assert pure-text`" warns you if it finds anything other than position-independent code, such as relocatable text, but not if it finds writable data.

Example: Bind—make the executable file `a.out`.

```
demo$ f77 teslib.f libfac.so.1.1
teslib.f:
 MAIN:
demo$ ■
```

Example: Run.

```
demo$ a.out
6
12.0000
demo$ ■
```

Inspect `a.out`for use of shared libraries. The `file` command shows if `a.out` is a dynamically linked executable — programs that use shared libraries are completely link-edited while they are executed, that is, dynamically.

Example: Use the `file` command to see if `a.out` is dynamically linked.

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo$ file a.out
a.out SPARC demand paged dynamically linked
      executable not stripped
demo$ ■
```

Sample Create of a Dynamic Library in 1.x (continued)

The `ldd` command shows that `a.out` uses some shared libraries, including `libfac.so.1` and `libc` (included by default by `f77`). It also shows exactly which files on the system will be used for these libraries.

Example: Use the `ldd` command to see if `a.out` uses shared libraries.

```
demo$ ldd a.out
        libfac.so.1.1
       -lF77.2 => /set/lang/3.0.1/lang/buildbin/4.x/libF77.so.2.0
        -lc.1 => /usr/lib/libc.so.1.6
demo$ ■
```

Your path may vary.

Dynamic Library for Exporting Initialized Data

Exported data means data in a shared library that is available for direct reference by programs using the library. For FORTRAN, exported initialized data is in `COMMON` statements and in `BLOCK DATA` routines.

In Solaris 1.x, if the data are assigned initial values in the library, then this set of data must be identified for the link editor by placing the data (and *only* the data) in a special random archive library with the `.sa` suffix. No such step is needed in Solaris 2.x.

Solaris 1.x

To create a dynamic library *that allows using initialized data*, do the following:

1. **Segregate the initializing declarations into `BLOCK DATA` routines.**
2. **Put them in separate source files.**
3. **Create a static archive library (a `.sa` file) composed of only those routines.**

   You *must* include these modules in the `.so` file.

4. **Use `ranlib` to incorporate a symbol table into this `.sa` archive library.**

Note – The above steps are for Solaris 1.x only. In Solaris 2.x it is all automatic.

Sample Create of a Dynamic Library—Export Initialized Data

Solaris 1.x

Example: Create dynamic library—allow exporting of initialized data.

```
demo$ cat Blkgrp.f
* Blkgrp.f -- Block Data for Shared Library
      blockdata blkgrp
      common / grp / a, b, c
      data a, b, c / 3*9.9 /
      end
demo$ cat PrintGrp.f
* PrintGrp.f -- Subroutine for Shared Library
      subroutine printgrp
      common / grp / a, b, c
      write( *, '(3f4.1)' ) a, b, c
      return
      end
demo$ cat ReadGrp.f
* ReadGrp.f -- Subroutine for Shared Library
      subroutine readgrp
      common / grp / a, b, c
      read( *, * ) a, b, c
      return
      end
demo$ cat TesSharMain.f
* TesSharMain.f -- Test Shared Library
      common / grp / a, b, c
      a = 1.0
      b = 2.0
      call printgrp
      end
demo$ ■
```

Sample Create of a Dynamic Library—Export Initialized Data (continued)

Example: Source that this *does* export initialized data.

```
demo$ cat ZapGrp.f
* ZapGrp.f -- Subroutine for Shared Library
      subroutine zapgrp
      common / grp / a, b, c
      a = 0.0
      b = 0.0
      c = 0.0
      return
      end
demo$ ■
```

Example: Use `-pic` on: `blkgrp.f`, `printgrp.f`, `readgrp.f`, `zapgrp.f`.

```
demo$ f77 -c -pic -silent *.f
demo$ ■
```

Example: Create the `.sa` file, then run `ranlib` on it.

```
demo$ ar cr libblkgrp.sa.1.1 Blkgrp.o
demo$ ranlib libblkgrp.sa.1.1
demo$ ■
```

Create a shared library `.so` file with the same version number as the `.sa` file. For the dynamic loader, the `.sa` and `.so` files must match exactly in name and version number.

Example: Create a shared library.

```
demo$ ld -o libblkgrp.so.1.1 -assert pure-text \
      PrintGrp.o ReadGrp.o ZapGrp.o Blkgrp.o
demo$ ■
```

Example: Bind.

```
demo$ f77 TesSharMain.o libblkgrp.so.1.1
demo$ ■
```

Example: Run.

```
demo$ a.out
 1.0 2.0 9.9
demo$ ■
```

Sample Create of a Dynamic Library—Export Initialized Data (continued)

Inspect the `a.out` file for use of shared libraries. The `file` command shows that `a.out` is a dynamically linked executable — programs that use shared libraries are completely link-edited while they are executed, that is, dynamically.

Example: Use the `file` command to see if `a.out` is dynamically linked.

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo$ file a.out
a.out: SPARC demand paged dynamically linked
     executable not stripped
demo$ ■
```

The `ldd` command shows that `a.out` uses two shared libraries, `libfac.so.1.1` and `libc` (included by default by `f77`.) It also shows exactly which files on the system will be used for these libraries.

Example: Use the `ldd` command to see if `a.out` uses shared libraries.

```
demo$ ldd a.out
     libblkgrp.so.1.1
     -lF77.2 => /set/lang/2.0/lang/buildbin/4.x/libF77.so.2.0
     -lc.0 => /usr/lib/libc.so.0.10
demo$ ■
```

7.5 Libraries Provided

Several libraries are installed with the compiler, including the following:

Table 7-2 Some Major Libraries Provided

Library	File	Options Needed
f77 functions, nonmath	`libF77`	*None*
f77 functions, nonmath, safe for multi-threaded code	`libF77_mt`	`-parallel`, etc.
f77 math library	`libM77`	*None*
VMS library	`libV77`	`-lV77`
Library used if linking Pascal, Fortran, and C objects	`libpfc`	*None*
Library of Sun math functions	`libsunmath`	*None*
POSIX bindings	`libFposix`	`-lFposix`
POSIX bindings that do extra runtime checking	`libFposix_c`	`-lFposix_c`

XView bindings and Xlib bindings for the X11 interface	`libFxview`	`-lFxview`
		`-lxview`
		`-lX11`

VMS Library

The `libV77` library is the VMS library. The VMS library contains the following special VMS routines.

`date`, `idate`, `secnds`, `time`, `ran`, `mvbits`

- To use any of these routines, include the `-lV77` option.
- For `idate` and `time`, there is a conflict between the VMS version and the version that traditionally is available on UNIX operating systems. If you use the `-lV77` option, you get the VMS compatible versions of the `idate` and `time` routines.

Read the *FORTRAN Reference Manual* for details about these routines.

7

POSIX Library

There are two versions of POSIX bindings provided.

- `libFposix`, which is just the bindings
- `libFposix_c`, which does some runtime checking to make sure you're passing correct handles.

If you pass bad handles:

- `libFposix_c` returns an error code (`ENOHANDLE`)
- `libFposix` dies with a segmentation fault

Of course, the checking costs, and `libFposix_c` is several times slower.

Both POSIX libraries come in static and dynamic flavors.

Which POSIX

The POSIX bindings provided are for IEEE Standard 1003.9-1992.

IEEE 1003.9 is a binding of 1003.1-1990 to FORTRAN (X3.8-1978).

POSIX.1 Documents:

- ISO / IEC 9945-1:1990
- IEEE Standard 1003.1-1990
- IEEE Order number SH13680
- IEEE CS Catalog number 1019

Those interested in knowing precisely what POSIX is probably need both the 1003.9 and the POSIX.1 documents.

For Further Information

Copies of the IEEE and ISO POSIX.1 Standard (ISO 9945-1:1990, also known as IEEE Standard 1003.1-1990) can be obtained from:

- Continental US

  Computer Society: +1 (714) 821 8380 (Ask for Customer Service)
  or IEEE Publication Sales +1 (800) 678-IEEE

- Canada

  IEEE Canada, +1 (908) 981-1393
  7071 Yonge St.
  Thornhill, Ontario L3T 2A6
  Canada.

- Outside Continental US or via paper

  IEEE Service Center+1 (800) 678-IEEE
  445 Hoes Lane, PO Box 1331
  Piscataway, NJ 08855-1331

  or

  IEEE Computer Society+1 (714) 821 8380
  10662 Los Vaqueros Circle, Fax +1 (714) 821 4010
  PO Box 3014
  Los Alamitos Ca. 90720-3014

- Europe

  IEEE Computer Society, +32 2 770 2198
  Jacques Kevers, Fax +32 2 770 8505
  13, Ave de l'Aquilon
  B-1200
  Brussels, Belgium.

- Asia

  IEEE Computer Society+81 33 408 3118
  Ms. Kyoko Mikami, Fax +81 33 408 3553
  Ooshima Building
  2-19-1 Minami Aoyma
  Minato-Ku, Tokyo 107
  Japan

7

Debugging 8

This chapter is organized into the following sections.

Global Program Checking (-Xlist)	*page 139*
Special Compiler Options (-C, -u, -U, -V, -xld)	*page 153*
The Debugger (dbx)	*page 155*
Debugging Parallelized Code	*page 168*
Compiler Messages in Listing (error)	*page 168*

8.1 Global Program Checking (`-Xlist`)

Purpose—Checking across routines helps find various kinds of bugs.

With `-Xlist`, `f77` reports errors of alignment, agreement in number and type for arguments, common blocks, parameters, plus many other kinds of errors (details follow).

It also makes a listing and a cross reference table; combinations and variations of these are available using suboptions. An example follows.

Example: *Errors only*—Use `-XlistE` to show *errors only.*

-XlistE

```
demo$ f77 -XlistE -silent Repeat.f
demo$ cat Repeat.lst
FILE  "Repeat.f"
program  repeat
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer*4
                 See: "Repeat.f" line #14
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #317:  variable "pn1" referenced as integer*4 across
               repeat/nwfrk//prnok in line #21 but set as real by repeat in
                 line #2
subroutine  subr1
    10              CALL subr1 ( x * 0.5 )
                               ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #3
subroutine  nwfrk
    17             PRINT *, prnok ( ix ), fork ( )
                                      ^
**** ERR  #418:  argument "ix" is integer*4, but dummy argument is real
                 See: "Repeat.f" line #20
subroutine  unreach_sub
    24           SUBROUTINE unreach_sub()
                                       ^
**** WAR  #338:  subroutine "unreach_sub" isn't called from program

Date:     Wed Feb 23 10:40:32 1994
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo$  ■
```

Errors in General

Global program checking can do the following:

- Enforce type checking rules of FORTRAN more stringently than usual, especially between separately compiled routines.
- Enforce some portability restrictions needed to move programs between different machines and/or operating systems
- Detect legal constructions that are nevertheless wasteful or error-prone
- Reveal other bugs and obscurities

Details

More particularly, global cross checking reports problems such as:

- Interface problems
  - Checking number and type of dummy and actual arguments
  - Checking type of function values
  - Checking possible conflicts of incorrect usage of data types in common blocks of different subprograms
- Usage problems
  - Function used as a subroutine or subroutine used as a function
  - Declared but unused functions, subroutines, variables, and labels
  - Referenced but not declared functions, subroutines, variables, and labels
  - Usage of unset variables
  - Unreachable statements
  - Implicit type variables
  - Inconsistency of the named common block lengths, names, and layouts
- Syntax problems—syntax errors found in a FORTRAN program
- Portability problems—codes that do not conform to ANSI FORTRAN, if the appropriate option is used

Using Global Program Checking

To cross check the named source files, use `-Xlist` on the command line.

Example: Compile three files for global program checking.

```
demo$ f77 -Xlist  any1.f  any2.f  any3.f
```

In the above example, `f77` does the following:

- Saves the output in the file `any1.lst`
- Compiles and links the program if there are no errors

Example: Compile *all* FORTRAN files for global program checking.

```
demo$ f77 -Xlist  *.f
```

Terminal Output

To display directly to the terminal, rename the output file to `/dev/tty`.

Example: Display to terminal.

```
demo$ f77 -Xlisto /dev/tty  any1.f
```

See *-Xlisto name*, on page 150.

The Default Output Features

The simple `-Xlist` option (as shown in the example above) provides a combination of features available for output. That is, with no other `-Xlist` options on the f77 command line, the plain, simple `-Xlist` option provides the following:

- The output file has the same name as the first file, with a `.lst` extension.
- The output content includes:
  - A line-numbered source listing (Default)
  - Error messages (embedded in listing) for inconsistencies across routines
  - Cross reference table of the identifiers (Default)
  - Pagination at 66 lines per page, 79 columns per line (Defaults)
  - No call graph (Default)
  - No expansion of include files (Default)

Analysis Files (`.fln` *Files)*

`f77` stores results of local cross checking analysis for source files into files with a `.fln` suffix. It usually uses the source directory. The files may be clutter.

Work around 1: Delete the files from time to time.

```
demo$ rm *.fln
```

Work around 2: Put the files into, say, `/tmp`. See `-Xlistfln`*dir*, page 149.

```
demo$ f77 -Xlistfln/tmp *.f
```

Example: Using `-Xlist`—A program with inconsistencies between routines.

`Repeat.f`

```
demo$ cat Repeat.f
      PROGRAM repeat
        pn1 = REAL( LOC ( rp1 ) )
        CALL subr1 ( pn1 )
        CALL nwfrk ( pn1 )
        PRINT *, pn1
      END ! PROGRAM repeat

      SUBROUTINE subr1 ( x )
        IF ( x .GT. 1.0 ) THEN
         CALL subr1 ( x * 0.5 )
        END IF
      END

      SUBROUTINE nwfrk( ix )
        EXTERNAL fork
        INTEGER prnok, fork
        PRINT *, prnok ( ix ), fork ( )
      END

      INTEGER FUNCTION prnok ( x )
        prnok = INT ( x ) + LOC(x)
      END

      SUBROUTINE unreach_sub()
        CALL sleep(1)
      END
demo$ f77 -Xlist -silent Repeat.f
demo$ cat Repeat.lst                    <output on following pages>
```

Compile with `-Xlist`. →
List the `-Xlist` output file. →

Example: Output file for `-Xlist`.

`Repeat.lst`

Error messages are embedded in the source listing.

```
FILE  "Repeat.f"
     1           PROGRAM repeat
     2             pn1 = REAL( LOC ( rp1 ) )
     3             CALL subr1 ( pn1 )
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer*4
                 See: "Repeat.f" line #14
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #317:  variable "pn1" referenced as integer*4 across
              repeat/nwfrk//prnok in line #21 but set as real by repeat in
                 line #2
     5             PRINT *, pn1
     6           END ! PROGRAM repeat
     7
     8           SUBROUTINE subr1 ( x )
     9             IF ( x .GT. 1.0 ) THEN
    10              CALL subr1 ( x * 0.5 )
                             ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #3
    11             END IF
    12           END
    13
    14           SUBROUTINE nwfrk( ix )
    15             EXTERNAL fork
    16             INTEGER prnok, fork
    17             PRINT *, prnok ( ix ), fork ( )
                                    ^
**** ERR  #418:  argument "ix" is integer*4, but dummy argument is real
                 See: "Repeat.f" line #20
    18           END
    19
    20           INTEGER FUNCTION prnok ( x )
    21             prnok = INT ( x ) + LOC(x)
    22           END
    23
    24           SUBROUTINE unreach_sub()
                                     ^
**** WAR  #338:  subroutine "unreach_sub" isn't called from program
    25             CALL sleep(1)
    26           END
```

Output File: "`f77 -Xlist Repeat.f`" (*continued*)

`Repeat.lst` (*continued*)

Cross reference table

```
         C R O S S   R E F E R E N C E   T A B L E
  Source file:   Repeat.f
Legend:
D        Definition/Declaration
U        Simple use
M        Modified occurrence
A        Actual argument
C        Subroutine/Function call
I        Initialization: DATA or extended declaration
E        Occurrence in EQUIVALENCE
N        Occurrence in NAMELIST

         P R O G R A M   F O R M
 Program
 -------
repeat          <repeat>        D      1:D

 Functions and Subroutines
 -------------------------
fork     int*4  <nwfrk>        DC     15:D     16:D     17:C

int      intrinsic
                <prnok>         C     21:C

loc      intrinsic
                <repeat>        C      2:C
                <prnok>         C     21:C

nwfrk           <repeat>        C      4:C
                <nwfrk>         D     14:D

prnok    int*4  <nwfrk>        DC     16:D     17:C
                <prnok>        DM     20:D     21:M

real     intrinsic
                <repeat>        C      2:C

sleep           <unreach_sub>             C     25:C

subr1           <repeat>        C      3:C
                <subr1>        DC      8:D     10:C
unreach_sub     <unreach_sub>             D     24:D
```

Sample Interpretation:

The routine `nwfrk` → called in `repeat`, line 4
defined, line 14

Output File: "`f77 -Xlist Repeat.f`" (*continued*)

`Repeat.lst` (*continued*)

More of the cross reference table

```
 Variables and Arrays
 --------------------

ix       int*4  dummy
                <nwfrk>        DA     14:D     17:A

pn1      real*4 <repeat>      UMA      2:M      3:A      4:A      5:U

rp1      real*4 <repeat>        A      2:A

x        real*4 dummy
                <subr1>        DU      8:D      9:U     10:U
                <prnok>       DUA     20:D     21:A     21:U

------------------------------------------------------------------------

Date:     Tue Feb 22 13:15:39 1994
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo$ ■
```

Interpretation—In the above example, reading the cross reference table:

- `ix` is a 4-byte integer
  - Used as an argument in routine `nwfrk`
  - At line 14, declaration of argument
  - At line 17, used as an actual argument
- `pn1` is a 4-byte real in routine `repeat`
  - At line 2, modified
  - At line 3, argument
  - At line 4, argument
  - At line 5, used
- `rp1` is a 4-byte real in routine `repeat`
  - At line 2, argument
- `x` is a 4-byte real in routines `subr1` and `prnok`
  - In `subr1`, at line 8, defined; at lines 9 & 10 used
  - In `prnok`, at line 20, defined; at line 21, used as an argument

Suboptions for Global Checking Across Routines

The standard global cross checking option is `-Xlist` (with no suboption).

This shows the listing, errors, and cross reference table. For variations from this standard report, add one or more suboptions to the command line.

Suboption Syntax

Add suboptions according to the following rules:

- Append the suboption to `-Xlist`
- Put no space between the `-Xlist` and the suboption
- Put only one suboption per `-Xlist`

Combination Special and A La Carte Suboptions

Combine suboptions according to the following rules:

- The *combination special*: `-Xlist` (listing, errors, and cross reference table)
- The *a la carte* options are: `-XlistC`, `-XlistE`, `-XlistL`, and `-XlistX`.
- All other options are detail options, not *a la cart*, not *combination special.*
- Once you start ordering *a la carte*, the three parts of the *combination special* are cancelled, and you get only what you specify.

Example: Each of these two commands does the same thing.

```
demo$ f77  -Xlistc  -Xlist  any.f
demo$ f77  -Xlistc  any.f
```

Combination special or *a la carte* suboptions (with no other suboptions):

Type/Amount of Output	Option	Comment	Details
Errors, listing, cross reference table	`-Xlist`	No suboptions	*page 142*
Errors	`-XlistE`	By itself, does not show listing or cross reference table	*page 148*
Errors and *listing*	`-XlistL`	By itself, does not show cross reference table	*page 149*
Errors and *cross reference table*	`-XlistX`	By itself, does not show listing	*page 151*
Errors and *call graph*	`-XlistC`	By itself, does not show listing or cross reference table	*page 148*

Summary of `-Xlist` Suboptions

Option	Action	Details
`-Xlist` *(no suboption)*	Errors, listing, and cross reference table.	*page 147*
`-Xlist`**`c`**	Call graphs (and errors).	*page 148*
`-Xlist`**`E`**	Errors.	*page 148*
`-Xlist`**`err[`***`nnn`***`]`**	Suppress error *nnn* in the verification report.	*page 148*
`-Xlist`**`f`**	Fast output.	*page 149*
`-Xlist`**`fln`***`dir`*	The `.fln` files directory.	*page 149*
`-Xlist`**`I`**	Include files.	*page 149*
`-Xlist`**`L`**	Listing (and errors).	*page 149*
`-Xlist`**`l`***`n`*	Page breaks.	*page 149*
`-Xlist`**`o`** *name*	Rename the `-Xlist` output report file.	*page 150*
`-Xlist`**`s`**	Suppress unreferenced identifiers from the cross reference table.	*page 150*
`-Xlist`**`v`***`n`*	Show different amounts of semantic information.	*page 150*
`-Xlist`**`w[`*****`nnn`*****`]`**	Width of output lines.	*page 151*
`-Xlist`**`war[`*****`nnn`*****`]`**	Suppress warning *nnn* in the report.	*page 151*
`-Xlist`**`X`**	Cross reference table (and errors).	*page 151*

Details of `-Xlist` *Suboptions*

**`-Xlist``c`** Call graphs (and errors). Show cross check errors and call graphs. This suboption by itself does not show a listing or a cross reference. Produce the call graph in a planned tree form using printable characters. If some subroutines are not called from `MAIN`, more than one graph is shown. Each `BLOCKDATA` is printed separately with no connection to `MAIN`. Default: Do not show the call graph.

`-Xlist``E` Global cross check errors. Show cross routine errors. This suboption by itself does not show a listing or a cross reference.

**`-Xlist``err[`***`nnn`***`]`** Suppress error *nnn* in the verification report. This is useful if you want a cross reference or a listing without the error messages. It is also useful if you do consider certain practices to be real errors. To suppress more than one error, use this option repeatedly. Example: `-Xlisterr338` suppresses *error* message 338. If *nnn* is not specified, then suppress all error messages.

`-Xlistf` Fast output. Produce source file listings and cross checking, verify sources much faster, but generate no object files. Default: Generate object files.

`-Xlistfln`*dir* `.fln` directory *dir*. Put the `.fln` files into the *dir* directory.

The *dir* directory must already exist.

Default: The source directory.

`-XlistI` Include files. List and cross check the include files.

If `-XlistI` is the only `-Xlist` option/suboption used, then you get the standard `-Xlist` output of a line numbered listing, error messages, and a cross reference table—but include files are shown or scanned, as appropriate.

- Listing

  If the listing is not suppressed, then the include files are listed in place. Files are listed as often as they are included. The following are all listed:
  - Source files
  - `#include` files
  - `INCLUDE` files

- Cross Reference Table

  If the cross reference table is not suppressed. The following are all scanned while generating the cross reference table:
  - Source files
  - `#include` files
  - `INCLUDE` files

Default: No include files.

`-XlistL` Listing (and errors). Show cross check errors and listing. This suboption by itself does not show a cross reference. Default: Show listing, cross reference.

`-Xlistl`*n* Page breaks. Set the page length for pagination to *n* lines. That is the letter *ell* for length, not the digit *one*. For example, `-Xlistl45` sets the page length to 45 lines. Default: 66.

No Page Breaks: The `-Xlistl0` {that is a *zero*, not a letter *oh*} option shows listings and cross reference with no page breaks (easier for on-screen viewing).

`-Xlisto` *name*

Rename the `-Xlist` output report file. The space between `o` and *name* is required. Output is then to the *name*`.lst` file.

To display directly to the terminal: `-Xlisto /dev/tty`

`-Xlists`

Suppress unreferenced identifiers from cross reference table.

If identifiers are

- Defined in include files
- Not referenced in the source files

then they are not shown in the cross reference table.

This suboption has no effect if the suboption `-XlistI` is used.

Default: Do not show the occurrences in `#include` or `INCLUDE` files.

`-Xlistv`*n*

n=1,2,3,4 Set level of checking strictness. Default: 2.

`-Xlistv1`

Show cross checked information of all names in summary form only, no line numbers. Set lowest level of checking strictness (syntax errors only).

`-Xlistv2`

Show cross checked information with summaries and line numbers. Set normal level of checking strictness (plus: argument inconsistency errors and variable usage errors). Default for `-Xlist` is `-Xlistv2`.

`-Xlistv3`

Show cross checking with summaries and line numbers. Additionally to `-Xlistv2`, show common block maps. Set high level of checking strictness (plus: errors caused by incorrect usage of data types in common blocks in different subprograms).

`-Xlistv4`

Show cross checking with summaries and line numbers. Additionally to `-Xlistv2`, show common block maps and equivalence block maps. Set top level of checking strictness (maximum error detection).

-Xlistw[*nnn*] Width of output lines. Set the output line width to *n* columns. For example, -Xlistw132 sets the page width to 132 columns. Default: 79.

-Xlistwar[*nnn*] Suppress warning *nnn* in the report. If *nnn* is not specified, then all warning messages will be suppressed from printing. To suppress more than one, but not all warnings, use this option repeatedly. For example, -Xlistwar338 suppresses *warning* message 338.

-XlistX Cross reference table (and errors). Show cross checking errors and cross reference. This suboption by itself does not show a listing.

The cross reference table shows information about each identifier:

- Is it an argument?
- Does it appear in a COMMON or EQUIVALENCE declaration?
- Is it set or used?

Example: Use -Xlistwar*nnn* to suppress two specific warnings.

-Xlistwar*n*

Suppress specific warnings

```
demo$ f77 -Xlistwar338  -Xlistwar348 -XlistE -silent Repeat.f
demo$ cat Repeat.lst
FILE  "Repeat.f"
program  repeat
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer*4
                 See: "Repeat.f" line #14
     4             CALL nwfrk ( pn1 )
                                ^
**** ERR  #317:  variable "pn1" referenced as integer*4 across
              repeat/nwfrk//prnok in line #21 but set as real by repeat in
                 line #2
subroutine  nwfrk
    17             PRINT *, prnok ( ix ), fork ( )
                                     ^
**** ERR  #418:  argument "ix" is integer*4, but dummy argument is real
                 See: "Repeat.f" line #20

Date:     Wed Feb 23 10:40:32 1994
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo$  ■
```

Some warnings that are popular to suppress are: 314, 315, 320, 357.

Example: Explain a mysterious message. Find a type mismatch.

ShoGetc.f

```
demo$ cat ShoGetc.f
     CHARACTER*1 c
     i = getc(c)
     END
demo$ f77 -silent ShoGetc.f
demo$ a.out
Z
 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
demo$ f77 -XlistE -silent ShoGetc.f
demo$ cat ShoGetc.lst
FILE  "ShoGetc.f"
program  MAIN
     2          i = getc(c)
                ^
**** WAR  #320:  variable "i" set but never referenced
     2          i = getc(c)
                       ^
**** ERR  #412:  function "getc" used as real but declared as
integer*4
     2          i = getc(c)
                        ^
**** WAR  #320:  variable "c" set but never referenced

Date:     Fri Mar  4 12:13:11 1994
Files:         2 (Sources: 1; libraries: 1)
Lines:         3 (Sources: 3; Library subprograms:1)
Routines:      1 (MAIN: 1)
Messages:      3 (Errors: 1; Warnings: 2)
demo$ cat ShoGetc.f
     CHARACTER*1 c
     INTEGER c
     i = getc(c)
     END
demo$ f77 -silent ShoGetc.f
demo$ a.out
Z
demo$  ■
```

The problem: Why this message? →

The debugging: Use `-Xlist`. List the output.

Here is the error. → Our default typing of `getc` is not consistent with the FORTRAN library.

`f77` was given special information about the FORTRAN library—that is how `f77` knows that `getc` is integer.

The solution: → Make c an integer.

No more mysterious message.

8.2 Special Compiler Options (-C, -u, -U, -V, -xld)

The compiler options -C, -u, -U -V, and -xld are useful for debugging. They check subscripts, spot undeclared variables, show stages of the compile/link sequence, versions of software, and compile D debug statements.

Solaris 2.3 and later: New linker debugging aids, see ld(1), or try -Qoption ld -Dhelp.

Subscript Bounds

-C If you compile with -C, then f77 checks at runtime for out-of-bounds on each array subscript. This helps catch some causes of the dreaded segmentation fault.

Example: Index out of range.

```
demo$ cat indrange.f
      REAL a(10,10)
      k = 11
      a(k,2) = 1.0
      END
demo$ f77 -C -silent indrange.f
demo$ a.out
 Subscript out of range on file indrange.f, line 3, procedure MAIN.
 Subscript number 1 has value 11 in array a.
 Abort (core dumped)
demo$ ■
```

Undeclared Variable Types

-u The -u compiler option causes all variables to be initially identified as undeclared, so that an error is flagged for variables that are not explicitly declared. The -u flag is useful for discovering mistyped variables. If -u is set, all variables are treated as undeclared until explicitly declared. Use of an undeclared variable is accompanied by an error message.

Case-sensitive Variable Recognition

-U If you debug FORTRAN programs that use other languages, it is generally safer to compile with the `-U` option to get case-sensitive variable recognition.

With the `-U` option, `f77` does *not* convert uppercase letters to lowercase, but leaves them in the original case. The default is to convert to lowercase except within character-string constants.

Note – If you are not consistent concerning the case of your variables, this `-U` option could also cause serious problems: if you sometimes type `Delta` and other times type `DELTA` or `delta`, then the `-U` leaves them as different symbols. This may not be what you intend.

Version Checking

You can check the versions of the various components of the compiler. This can be useful after installing a patch, or after two or more people each did part of an installation.

`version`

The "`version f77`" command displays the version ID of the compiler driver.

-V The `-V` compiler option displays the name and version ID of each phase of the compiler. This can be useful in tracking the origin of ambiguous error messages and in reporting compiler failures.

D Comment Line Debug Statements

-xld The -xld flag causes the compiler to compile statements that have a D or a d in column one. Without it, they are comments. See Section 3.7, "Compiler Directives," for details on -xl[d].

Example: Compile with and without -xld.

forxld.f

With -xld, this prints I and X. Without -xld, it does not print them.

```
      REAL A(5) / 5.0, 6.0, 7.0, 8.0, 9.0 /
      DO I = 1, 5
           X = A(I)**2
D          PRINT *, I, X
      END DO
      PRINT *, 'done'
      END
```

*8.3 The Debugger (*dbx*)*

This section is organized as follows:

Sample Program for Debugging		*page 156*
Sample dbx Session	*(example)*	*page 157*
Segmentation Fault—Finding the Line Number	*(example)*	*page 159*
Exception—Finding the Line Number	*(example)*	*page 161*
Bus Error—Finding the Line Number	*(example)*	*page 162*
Trace of Calls	*(example)*	*page 163*
Print Arrays	*(example)*	*page 164*
Print Array Slices	*(example)*	*page 165*
Miscellaneous Tips		*page 166*
Main Features of the Debugger		*page 167*

This section *introduces* some dbx features likely to be used with Fortran. Use it as a *quick start* for debugging Fortran.

Note – Before you use the debugger, you must install the appropriate Tools package—read *Installing SunSoft Developer Products on Solaris* for details.

8

Sample Program for Debugging

The following program, consisting of files `a1.f`, `a2.f`, `a3.f`, with bug, is used in several examples of debugging.

Example: Main for debugging.

a1.f

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

Example: Function for debugging.

a3.f

```
      REAL FUNCTION determinant ( a )
      REAL a(2,2)
      determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
      RETURN
      END
```

Example: Subroutine for debugging.

a2.f

```
      SUBROUTINE mkidentity ( array, m )
      REAL array(m,m)
      DO 90 i = 1, m
           DO 20 j = 1, m
               IF ( i .EQ. j ) THEN
                 array(i,j) = 1.
               ELSE
                 array(i,j) = 0.
               END IF
20         CONTINUE
90    CONTINUE
      RETURN
      END
```

Sample dbx *Session*

The following examples use the sample program above.

- *Compile*—To use dbx or debugger, compile and link with the -g flag. You can do this in one step or two, as shown in the examples below.

  Example Compile and link *in one step*, with -g.

```
demo$ f77 -o silly -g a1.f a2.f a3.f
```

  Example: Compile and link *in separate steps*.

```
demo$ f77 -c -g a1.f a2.f a3.f
demo$ f77 -g -o silly a1.o a2.o a3.o {-g needed in Solaris 1.x, but not in 2.x}
```

- *Start* dbx—To start dbx, type dbx and the name of your executable file.

  Example: Start dbx on the executable named silly.

```
demo$ dbx silly
Reading symbolic information…
(dbx) ■
```

- *Quit* dbx—To quit dbx, enter the quit command.

  Example: Quit dbx.

```
(dbx)quit                                  {Skip this for now so you can do the next steps.}
demo$ ■
```

- *Breakpoint*—To set a breakpoint, wait for dbx prompt; then type "stop in *subnam*", where *subnam* names a subroutine, function, or block data subprogram.

  Example: A way to stop at the first executable statement in a main program.

The "MAIN" must be ***upper***case. →

```
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) ■
```

  Although "MAIN" must be upper case, in general, the "*subnam*" can be upper or lower case. See *-U*, page 154.

- *Run Program*—To run the program from `dbx`, enter the `run` command. It runs the program in the executable files named when you started `dbx`.

  Example: Run the program from within `dbx`.

```
(dbx) run
Running: silly
stopped in MAIN at line 3 in file "a1.f"
    3         call mkidentity( twobytwo, n )
(dbx) ■
```

  When the breakpoint is reached, `dbx` displays a message showing where it stopped, in this case at line 3 of the `a1.f` file.

- *Print*—To print a value, enter the `print` command.

  Example: Print value of `n`.

```
(dbx) print n
n = 2
(dbx) ■
```

  Example: Print the matrix `twobytwo` (format may vary with release).

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
(dbx) ■
```

  Example: Print the matrix `array`.

The error message details may vary with the release, and of course with any translation.

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx) ■
```

  The print fails because `array` is not defined here—only in `mkidentity`.

- *Next Line*—To advance execution to the next line, enter the `next` command.

  Example: Advance execution to the next line.

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4           print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo$ ■
```

  The `next` command executes the current source line, then stops at the next line. It counts subprogram calls as single statements.

  Compare `next` with `step`. The `step` command executes the next source line, or the next step into a subprogram, and so forth. In general, if the next executable source statement is a subroutine or function call, then

  - `step` sets a breakpoint at the first source statement of the subprogram.
  - `next` sets the breakpoint at the first source statement after the call but still in the calling program.

Segmentation Fault—Finding the Line Number

If a program gets a *segmentation fault* (`SIGSEGV`), it referenced a memory address outside of the memory available to it.

Some Causes of `SIGSEGV`

There are many possible causes, but the most common is an array index being outside the declared range. The most frequent causes are the following:

- The name of an array index is misspelled
- The calling routine has a `REAL` argument; called routine has it as `INTEGER`
- An array index is miscalculated
- The calling routine calls with fewer arguments than required
- A pointer is used before it is defined

There are several ways to locate the offending source line, but any of the following ways can be helpful:

- Recompile with the `-Xlist` option to get global program checking
- Recompile with the `-C` subscript checking option. See "Subscript Bounds."
- Use `dbx` to find the source code line where a segmentation fault occurred

Example: Program to generate a segmentation fault.

```
demo 4% cat WhereSEGV.f
      INTEGER a(5)
      j = 2000000
      DO 9 i = 1,5
           a(j) = (i * 10)
9     CONTINUE
      PRINT *, a
      END
demo 5% ■
```

Example: Use `-C` to locate a segmentation fault.

```
demo 5% f77 -C -silent WhereSEGV.f
demo 6% a.out
Subscript out of range on file WhereSEGV.f, line 4, procedure
MAIN.
Attempt to access the 2000000-th element of variable a.
Abort (core dumped)
demo 7% ■
```

Example: Use `dbx` to get the line number of a segmentation fault.

```
demo 5% f77 -g -silent WhereSEGV.f
demo 6% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation fault (core dumped)
demo 7% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx) ■
```

Exception—Finding the Line Number

If a program gets an exception, there are many possible causes, so one approach is to find the line number (in the source program) where the exception occured. Then look for clues.

Example: Find where an exception occurred.

`WhereExcept.f`

You can find the source code line number where a floating-point exception occurred by using the `ieee_handler` routine with either `dbx` or `debugger`.

Note the "`catch FPE`" `dbx` command →

```
      EXTERNAL myhandler                                ! Main
      INTEGER ieeer, ieee_handler, myhandler
      REAL r/14.2/, s/0.0/
      ieeer = ieee_handler('set', 'all', myhandler)
      PRINT *, r/s
      END
      INTEGER FUNCTION myhandler(sig, code, context) ! Handler
*     {This handler is OK in SunOS 4.X/5.0 since it just aborts.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo$ f77 -g -silent WhereExcept.f
demo$ dbx a.out
Reading symbolic information for a.out
(dbx) catch FPE
(dbx) run
Running: a.out
signal FPE (floating point divide by zero)
     in MAIN at line 5 in file "WhereExcept.f"
    5          PRINT *, r/s
(dbx) ■
```

Bus Error—Finding the Line Number

If a program gets a bus error (SIGBUS), it usually has had some problem with misaligned data. The address may well be valid, whereas with SIGSEGV the address is invalid. Some possible causes of SIGBUS are:

- Misaligned data
- Using a pointer that is not defined, or that is incorrectly defined

Example: Program to generate a bus error (SIGBUS).

The code is artificial, of course, but it does generate a SIGBUS.

The caller assumes the argument is aligned as a character, bytes 2-5.

The called routine assumes it is aligned as an integer.

```
demo$ cat WhereSIGBUS.f
      character*1 c(5)
      call sub(c(2))
      end
      subroutine sub(i)
      print *,i
      end
demo$ f77 -C -silent WhereSIGBUS.f
demo$ a.out
*** TERMINATING a.out
*** Received signal 10 (SIGBUS)
Bus Error (core dumped)
demo$ ■
```

Example: Recompile with the -Xlist to locate a bus error (SIGBUS).

```
demo 5% f77 -Xlist -silent WhereSIGBUS.f
demo 6% cat WhereSIGBUS.lst
WhereSIGBUS.f             Fri Jun 10 16:02:17 1994                    page 1
FILE  "WhereSIGBUS.f"
     1          character*1 c(5)
     2          call sub(c(2))
                           ^
**** ERR  #418:  argument "c" is character, but dummy argument is integer*4
                 See: "WhereSIGBUS.f" line #4
     2          call sub(c(2))
                           ^
**** ERR  #316:  array "c" may be referenced before set by sub in line #5
     3          end
     4          subroutine sub(i)
     5          print *,i
     6          end
< many lines omited>
(dbx) ■
```

Trace of Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that brought it there (a *stack trace*).

Example: Show the sequence of calls, starting at where the execution stopped.

ShowTrace.f is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

```
demo$ f77 -silent -g ShowTrace.f
demo$ a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Reading symbolic information for a.out
...
(dbx) run
Running: a.out
(process id 1089)
signal SEGV (no mapping at the fault address) in calcb at line
23 in file "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where
=>[1] calcb(v =  ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a =  ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx) ■
```

Note reverse order:
MAIN called calc
calc called calcb.

Execution stopped, line 23 → calcB called from calc, line 9 → calc called from MAIN, line 3 →

The `where` command shows where in the program flow execution stopped (how execution reached this point), that is, a *stack trace* of the called routines. This can be helpful, since you no longer get an *automatic* traceback, as bemoaned in the ode below.

> Ode To Traceback
>
> O blinding core! File of death!
> Alone like Abel's brother, Seth.
> The demise of process I cannot face
> Without the aid of stackish trace.
> To see what by you must needs be done,
> Please see Example Twenty-One.[1]
>
> © *Mateo Burtch, 1992*

1. Since `trace` be dead, or just not there, try `dbx`'s better `where`.
Seek not example twenty one, as it was cited just for fun.

Print Arrays

Example: dbx recognizes arrays. It can print arrays.

Arraysdbx.f

```
demo$ dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx) stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
     (1,1) 11
     (2,1) 21
     (3,1) 31
     (4,1) 41
     (1,2) 12
     (2,2) 22
     (3,2) 32
     (4,2) 42
     (1,3) 13
     (2,3) 23
     (3,3) 33
     (4,3) 43
     (1,4) 14
     (2,4) 24
     (3,4) 34
     (4,4) 44
(dbx) print IARR(2,3)
     iarr(2, 3) = 23  ← order of user-specified subscripts ok
(dbx) quit
demo$ ■
```

Print Array Slices

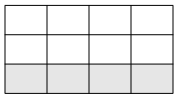
Example: dbx prints array *slices* if you specify which rows and columns.

ShoSli.f

This is one way of printing portions of large arrays.

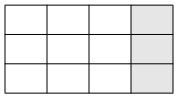
```
demo$ f77 -g -silent ShoSli.f
demo$ dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
     1          INTEGER*4  a(3,4), col, row
     2          DO row = 1,3
     3                  DO col = 1,4
     4                    a(row,col) = (row*10) + col
     5                  END DO
     6          END DO
     7          DO row = 1, 3
     8                   WRITE(*,'(4I3)') (a(row,col),col=1,4)
     9          END DO
     10         END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
     7           DO row = 1, 3
(dbx)■
```

Example: Print row 3.



```
(dbx) print a(3:3,1:4)
‘ShoSli‘MAIN‘a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)■
```

Example: Print column 4.



```
(dbx) print a(1:3,4:4)
‘ShoSli‘MAIN‘a(3:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)■
```

8

Miscellaneous Tips

The following tips and background concepts can help (in case you have not yet read all of *Debugging a Program*).

Current Procedure and File

During a debug session, `dbx` defines a procedure and a source file as *current*. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, "`stop at 5`" sets one of three different breakpoints, depending on whether the current file is `a1.f, a2.f`, or `a3.f`.

Uppercase Letters

If your program has uppercase letters in any identifiers, then `dbx` recognizes them. You do not need to give `dbx` any specific commands, as in some earlier versions.

If you are using the `debugger` program, rather than `dbx`, then whenever you supply arguments to the file and directory management commands, the arguments are case sensitive. This is true even if you have already set the "`dbxenv case insensitive`" environment attribute. These management commands are available in the command pane.

Debugging Optimized Programs

For debugging optimized programs, one way that many users find helpful is:

- Compile the main program with `-g` but with no `-O`*n*.
- Compile every other routine of the program with the appropriate `-O`*n*.
- Start the execution under `dbx`
- Use "`fix -g` *any*`.f`" (on the routine you want to debug, *but no* `-O`*n)*
- Use `continue` with that routine compiled.

Runtime Checking

The `dbx` *runtime checking* feature can be very helpful for standard C programs that use pointers. For standard FORTRAN programs, it is usually not helpful. The more common FORTRAN problem of an array index accessing outside of the array can be caught with `-C`; see "Subscript Bounds" on page 153.

Main Features of the Debugger

Be sure to read *Debugging a Program* for the following:

- The full range of features in the debugger
- The window- and mouse-based interface
- An appendix with more FORTRAN examples

Overview of dbx *Features Useful for FORTRAN*

The dbx program provides event management, process control, and data inspection. It allows you to watch what is happening during program execution. You can do such things as the following:

- *Fix* one routine, then *continue* executing without recompiling the others
- *Set watchpoints* to stop or trace if a specified item changes

Solaris 2.x

- *Collect data* for the performance-tuning Analyzer
- *Graphically monitor* variables, structures, and arrays—Data Inspector

- *Set breakpoints* (set places to halt in the program) at lines or in functions
- *Show values*—once halted, show or modify variables, arrays, structures, ...
- *Step* through program, one source line at a time (or one assembly line)
- *Trace* program flow (show sequence of calls taken)
- *Invoke procedures* in the program being debugged
- *Step* over or into function calls; step up and out of a function call
- *Run, stop, and continue* execution (at the next line or at some other line)
- *dbx-safe I/O* in the command window—Program Input/Output Window
- *Save and then replay* all or part of a debugging run
- *Stack*—Examine the call stack; move up and down the call stack
- *Program* scripts by embedded Korn shell

Solaris 2.x

- *Follow programs* as they `fork`(2) and `exec`(2)

8

8.4 Debugging Parallelized Code

The debug option conflicts with the parallelizing options.

The Problem

- If you compile a routine with `-g` and a parallelizing option, the compiler turns off the parallelizing option. The `-autopar`, `-explicitpar`, and `-parallel` options conflict with `-g` because `dbx` cannot do useful debugging on a routine compiled with `-parallel`, `-autopar`, or `-explicitpar`.

  Example: With `-g`, `f77` turns off `-parallel`.

```
demo$ f77 -O3 -g -parallel any.f
f77: Warning: -parallel conflicts with -g. -parallel turned off.
demo$
```

For solutions, see Section C.5, “Debugging Tips and Hints for Parallelized Code,” on page C-309.

8.5 Compiler Messages in Listing (`error`)

`error` inserts compiler diagnostics above the relevant line in the source file.

- This includes the standard compiler error and warning messages.
- This does *not* include the `-Xlist` error and warning messages
- This changes your source files.
- This does not work if the source files are in a read-only directory.

At the time the operating system is installed, `error` is included if you do a *Developer Install*, rather than an *End User Install*; it is also included if you install the package `SUNWbtool`.

Method

The `error` utility associates compiler error diagnostics with the offending source lines. It recognizes and categorizes diagnostics from a variety of source-language processors and *inserts them as comments in the appropriate source file* before the lines that caused the corresponding errors. This allows the programmer to read the source code along with its compiler diagnostics.

Usage of `error` *Utility*

Use `error` as follows (pass `stdout` and `stderr` from `f77` to `error`):
`sh`:

```
demo$ f77 any.f 2>&1 | error options
```

`csh`:

```
demo% f77 any.f |& error options
```

Options for `error`

The general form for using `error` with options is as follows:

error [**-n**] [**-s**] [**-q**] [**-v**] [**-t** *suffixlist*] [*filename*]

-n Do not touch any files. This sends all diagnostics to the standard output.

-q Query before changing each file. If there is no `-q` option, then it changes all files encountered during the compilation (except those files for discarded error messages).

-v After all files have been touched, invoke `vi` to edit all the touched files starting with the first one and position the cursor at the first diagnostic. If `vi` cannot be located in the standard places, try `emacs`, `ex`, or `ed`.

-s Print out statistics regarding error categorization.

-T Produce a terse form of messages. This is intended for standard output.

-t*suffixlist* Touch only files whose suffixes appear in *suffixlist*. The *suffixlist* is a dot-separated list and a * is acceptable as a wildcard. Example: Touch only files with the suffixes `.h`, `.f*`, or `.t`:

```
demo$ error -t '.h.f*.t'
```

-S Display the errors in the standard output as they are produced.

`filename` Read error messages from *filename* rather than from the standard input.

Description

The `error` utility examines each line of its input and does the following.

1. Determines the language processor that produced the message, the file name, and line number of the offending line.
2. Inserts the message in the form of a special comment *into the source file* immediately preceding the erroneous line. It changes source files.

If the source line of a diagnostic cannot be determined, the diagnostic is sent to the standard output without touching any files.

Easy Scanning with an Editor

The `error` utility inserts diagnostics in appropriate files after all input is read. The `-s` option allows previewing diagnostics before files are changed.

All diagnostics are inserted as one-line comments starting with the marker ### and ending with %%%. These markers make it easy for a text processor to:

- Locate such messages in a file
- Remove such messages from a file.

The line number of the offending line, along with the language processor that issued the message, appears in the comment line as well.

Piping/Redirecting

Pass both standard output and standard error from `f77` to the `error` utility.

Example: `csh`.

```
csh: demo% f77 myprog.f |& error -q
 sh: demo$ f77 myprog.f 2>&1 | error -q
```

In each shell, the above command compiles and redirects or pipes the standard output and standard error to the `error` program. Then `error`, in turn, processes these diagnostics and queries you before touching `myprog.f` and all other source files invoked from `myprog.f`.

Sample Use of the `error` *Utility*

Example: Sample program for showing use of the `error` utility.

`forerror.f`
(before compile)

This FORTRAN source program contains various syntax errors.

```
demo$ cat forerror.f
C    Sample program
C
     program test
     automatic x
     logical flag
     character*256 fname
     common /ioiflg/ ictl
     flag =.true.
     go to 10
     if (flag) then
10        ictl = 1
     else
          ictl = 0
     endif
     do 200 i = 0, MAXNUM
     call getenv(fname
     go to 200
     write (0, 2000) fname(:5)
200  continue
     endif
2000 format (' This is a test ", b)
     end
```

Compile the above program for `error` (csh):

```
demo$ f77 -ansi forerror.f |& error
```

Example: Source file changed by the `error` utility.

`forerror.f`
(after compile)

Note that the source file gets changed.

```
demo$ cat forerror.f
C###0 [Sunf77] ANSI extension: source line(s) in nonStandard
format%%%
C    Sample program
C
C###3 [Sunf77] ANSI extension: input contains lower case
letters%%%
     program test
C###4 [Sunf77] Warning: local variable "x" never used%%%
C###4 [Sunf77] ANSI extension: AUTOMATIC statement%%%
     automatic x
     logical flag
     character*256 fname
     common /ioiflg/ ictl
     flag =.true.
     go to 10
C###10 [Sunf77] Warning: statement cannot be reached%%%
     if (flag) then
C###11 [Sunf77] Warning: there is a branch to label 10 from
outside block%%%
10        ictl = 1
     else
          ictl = 0
     endif
     do 200 i = 0, MAXNUM
C###16 [Sunf77] Error: unclassifiable statement%%%
C###16 [Sunf77] Error: unbalanced parentheses, statement
skipped%%%
     call getenv(fname
     go to 200
C###18 [Sunf77] Warning: statement cannot be reached%%%
     write (0, 2000) fname(:5)
200  continue
C###20 [Sunf77] Error: endif out of place%%%
     endif
C###21 [Sunf77] Error: unclassifiable statement%%%
C###21 [Sunf77] Error: unbalanced quotes; closing quote
supplied%%%
C###21 [Sunf77] Error: unbalanced parentheses, statement
skipped%%%
2000 format (' This is a test ", b)
     end
demo$ ■
```

Floating Point 9

This chapter is organized into the following sections.

Summary	*page 173*
IEEE Solutions	*page 174*
The General Problems	*page 174*
IEEE Exceptions	*page 176*
IEEE Routines	*page 177*
Debugging IEEE Exceptions	*page 195*
Guidelines	*page 196*
Miscellaneous Examples	*page 197*

9.1 Summary

This chapter introduces floating-point problems and IEEE floating-point tools for solving those problems. It is for scientists and engineers who already understand floating-point arithmetic.

If you do not yet understand floating-point arithmetic, start by reading:

- *Numerical Computation Guide* (more explanations, examples, and details)
- *What Every Computer Scientist Should Know About Floating-point Arithmetic,*" by David Goldberg. It is in the AnswerBook system.

9.2 The General Problems

How can IEEE arithmetic help solve real problems? IEEE 754 standard floating-point arithmetic offers the user greater control over computation than is possible in any other type of floating point. In scientific research, there are many ways for errors to creep in.

- The model may be wrong.
- The algorithm may be numerically unstable (solving equations by inverting A^TA for example).
- The data may be ill-conditioned.
- The computer may be doing something astonishing, or at least unexpected.

It is nearly impossible to separate these error sources. Using library packages which have been approved by the numerical analysis community reduces the chance of there being a code error. Using good algorithms is another must. Using good computer arithmetic is the next obvious step.

The IEEE standard represents the work of many of the best arithmetic specialists in the world today. It was influenced by the mistakes of the past. It is, by construction, better than the arithmetic employed in the S/360 family, the VAX family, the CDC[1], CRAY, and UNIVAC[2] families (to name but a few). This is not because these vendors are not clever, but because the IEEE pundits came later and were able to evaluate the choices of the past, and their consequences. Does IEEE arithmetic solve all problems? No. But in general, the IEEE Standard makes it easier to write better numerical computation programs.

9.3 IEEE Solutions

IEEE arithmetic is a relatively new way of dealing with arithmetic operations where the result yields such problems as invalid, division by zero, overflow, underflow, or inexact. The big differences are in rounding, handling numbers near zero, and handling numbers near the machine maximum.

For rounding, IEEE arithmetic defaults to doing the intuitive thing, and closely corresponds with old arithmetic.

1. CDC is a registered trademark of Control Data Corporation.

2. UNIVAC is a registered trademark of UNISYS Corporation.

IEEE offers choices, which the expert can use to good effect, while old arithmetic did it just one way.

- What happens if we multiply two very large numbers with the same sign?
- Large numbers of different signs?
- Divide nonzero by zero?
- Divide zero by zero?

In old arithmetic, all these cases are the same. The program aborts on the spot; or in some very old machines, the computation proceeds, but with garbage. IEEE provides choices.

The *default* solution is to produce the following.

```
big*big = +Inf
big*(-)big = -Inf
num/0.0 = +Inf ← Where num > 0.0
num/0.0 = -Inf ← Where num < 0.0
0.0/0.0 = NaN  ← Not a Number
```

Above, `+Inf`, `-Inf`, and `NaN` are just introduced intuitively. More later.

Also an exception of one of the following kinds is raised.

- *Invalid* — Examples that yield invalid are 0.0 / 0.0, sqrt(-1.0), log(-37.8), …
- *Division by zero* — Examples that yield division by zero are 9.9 / 0.0, 1.0 / 0.0, …
- *Overflow* — Example with overflow: `MAXDOUBLE`+0.0000000000001e308
- *Underflow* — Example that yields underflow: `MINDOUBLE` * `MINDOUBLE`
- *Inexact* — Examples that yield inexact are 2.0 / 3.0, log(1.1), read in 0.1, … (no exact representation in binary for the precision involved)

There are various reasons to care about how all this works.

- If you don't understand what you are using, you may not like the results.
- Poor arithmetic can produce poor results. This cannot easily be distinguished from other causes of poor results.
- Switching everything to double precision is no panacea.

9

9.4 IEEE Exceptions

IEEE exception handling is the default on a SPARC processor. However, there is a difference between *detecting* a floating-point exception, and *generating a signal* for a floating-point exception (`SIGFPE`).

Detecting a Floating-point Exception

In accordance with the IEEE standard, two things happen when a floating-point exception occurs in the course of an operation.

- The handler returns a default result. For 0/0, return NaN as the result.
- A flag is set that an exception is raised. For 0/0, set “invalid operation” to 1.

Generating a Signal for a Floating-point Exception

The default on SPARC hardware systems is that they do *not* generate a *signal* for a floating-point exception. The assumption is that signals degrade performance, and that most users don’t care about most exceptions.

To generate a signal for a floating-point exception, you establish a signal handler. You use a predefined handler or write your own. See *Exception Handlers and ieee_handler()* on page 184.

Default Signal Handlers

By default, `f77` sets up some signal handlers, mostly for dealing with such things as a floating-point exception, interrupt, bus error, segmentation violation, or illegal instruction. *

Although you would not generally want to turn off this default behavior, it is possible to do so by setting the global C variable `f77_no_handlers` to 1, as shown in the following example.

Example: How to get *no* default signal handlers (set `f77_no_handlers` to 1.)

1. Create the following C program.

```
demo$ cat NoHandlers.c
     int  f77_no_handlers=1 ;
demo$
```

2. Compile it and save the `.o` file.

```
demo$ cc -c -o NoHand NoHandlers.c
demo$
```

3. Link the corresponding `.o` file into your executable file.

```
demo$ f77 NoHand.o Any.f
demo$
```

Otherwise (by default) it is 0. The effect is felt just before execution is transferred to the user's program so it does not make sense to set/unset it in the user's program.

Note that this variable is in the name space of the user's program, so do not use `f77_no_handlers` as the name of a variable anywhere else other than in the above C program.

9.5 IEEE Routines

The following interfaces help people use the functionality of IEEE arithmetic. These are mostly in the math library `libsunmath` and in several `.h` files.

- `ieee_flags`(3m)

  Control rounding direction. Control rounding precision. Query exception status. Clear exception status.

- `ieee_handler`(3m)

  Establish exception handler. Remove exception handler.

- `ieee_functions`(3m)

  List name and purpose of each IEEE function.

- `ieee_values`(3m)

  A list of functions that return special values.

- Other `libm` functions:
  - `ieee_retrospective`
  - `nonstandard_arithmetic`
  - `standard_arithmetic`

Many vendors support the IEEE standard. The SPARC processors conform to the IEEE standard in a combination of hardware and software support for different aspects.

The older Sun-4 uses the Weitek 1164/5, and the Sun-4/110 has that as an option.

The newer Sun-4 and the SPARCsystem series both use floating-point units with hardware square root. This is accessed if you compile with the `-cg89` option.

The newest SPARCsystem series uses new floating-point units, including SuperSPARC, with hardware integer multiply and divide instructions. These are accessed if you compile with the `-cg92` option.

The utility `fpversion` tells which floating-point hardware is installed. This utility runs on all Sun architectures. See `fpversion`(1), and read the *Numerical Computation Guide* for details. This replaces the older utility `fpuversion4`.

Flags and `ieee_flags()`

The `ieee_flags` function is used to query and clear exception status flags. It is part of the `libsunmath` shipped with SPARC operating systems.

It allows the programmer to do the following.

- Control rounding direction and rounding precision
- Check the status of the exception flags
- Clear exception status flags

The general form of a call to `ieee_flags` is as follows.

```
i = ieee_flags( action, mode, in, out )
```

- Each of the four arguments is a string.
- Input: `action`, *`mode`*, and *`in`*
- Output: *`out`* and *`i`*

- `ieee_flags` is an integer-valued function. Useful information is returned in *`i`*. Refer to the `man` page for `ieee_flags`(3m) for complete details.

Possible parameter values are shown below.

```
action:   get, set, clear, clearall
mode:     direction, precision, exception
in,out:   nearest, tozero, negative, positive,
          extended, double, single,
          inexact, division, underflow, overflow, invalid,
          all, common
```

The meanings of the possible values for `in` and `out` depend on the action and mode they are used with. These are summarized in the following table.

Table 9-1 `ieee_flags` Argument Meanings

Value of `in` **and** `out`	**Refers to**
nearest, tozero, negative, positive	Rounding direction
extended, double, single	Rounding precision
inexact, division, underflow, overflow, invalid	Exceptions
all	All 5 exceptions
common	Common exceptions: invalid, division, overflow

Note – These examples show only how to call the routines to get the information or set the behavior. They make no attempt to teach the numerical analysis that lets you know when to call them or what behavior to set.

Example: To determine what is the highest priority exception that has a flag raised, pass the input argument `in` as the null string.

```
      ieeer = ieee_flags( 'get', 'exception', '', out )
      PRINT *, out, ' flag raised'
```

Example: To determine if the `overflow` exception flag is raised, set the input argument `in` to `overflow`. On return, if `out` equals `overflow`, then the `overflow` exception flag is raised; otherwise it is not raised.

```
      ieeer = ieee_flags( 'get', 'exception', 'overflow', out )
      IF ( out.eq. 'overflow') PRINT *,'overflow flag raised'
```

Example: Clear the `invalid` exception.

```
      ieeer = ieee_flags( 'clear', 'exception', 'invalid', out )
```

Example: Clear all exceptions.

```
      ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

Example: Set rounding direction to zero.

```
      ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

Example: Set rounding precision to `double`.

```
      ieeer = ieee_flags( 'set', 'precision', 'double', out )
```

Turn Off All Warning Messages with `ieee_flags`

Use this only if you are certain you don't want to know about the unrequited exceptions. To do this, clear all accrued exceptions by putting a call to `ieee_flags()` just before your program exits.

Example: Clear all accrued exceptions with `ieee_flags()`.

```
      i = ieee_flags('clear', 'exception', 'all', out )
```

Detect an Exception with `ieee_flags`

Note – These examples show only how to call the routines to get the information. They make no attempt to teach the numerical analysis that lets you know when to call them and what to do with the information.

Example: Detect an exception using `ieee_flags`, and decode it.

Solaris 2.x

`DetExcFlg.F`

Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

```
#include "f77_floatingpoint.h"
      CHARACTER*16 out
      DOUBLE PRECISION d_max_subnormal, x
      INTEGER div, flgs, inv, inx, over, under

         x = d_max_subnormal() / 2.0                ! Cause underflow

         flgs=ieee_flags('get','exception','',out) ! Which are raised?

         inx   = and(rshift(flgs, fp_inexact)  , 1) ! Decode
         div   = and(rshift(flgs, fp_division) , 1)  ! the value
         under = and(rshift(flgs, fp_underflow), 1)   ! returned
         over  = and(rshift(flgs, fp_overflow) , 1)    ! by
         inv   = and(rshift(flgs, fp_invalid)  , 1)     ! ieee_flags

         PRINT *, "Highest priority exception is: ", out
         PRINT *, ' invalid  divide  overflo underflo inexact'
         PRINT '(5i8)', inv, div, over, under, inx
         PRINT *, '(1 = exception is raised; 0 = it is not)'
        i = ieee_flags('clear', 'exception', 'all', out) ! Clear all
         END
```

Example: Compile and run to detect an exception with `ieee_flags`.

```
demo% f77  -silent DetExcFlg.F
demo% a.out
 Highest priority exception is: underflow
  invalid  divide  overflo underflo inexact
       0       0       0       1       1
 (1 = exception is raised; 0 = it is not)
demo% ■
```

Detect All Five Exceptions with `ieee_flags`

Note – How to call, not when to call or what to do with the information.

Example: Detect all five exceptions using `ieee_flags`. Decode them.

`DetAllFlg.F`

Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

```
#include "f77_floatingpoint.h"
      CHARACTER*16 out
      DOUBLE PRECISION d_max_normal, d_max_subnormal, x, y /0.0/
      INTEGER div, flgs, inv, inx, over, under

      x = log( -37.8 )                        ! Cause invalid
      x = 3.14159 / y                         ! Cause division by zero
      x = d_max_subnormal() / 2.0             ! Cause underflow
      x = d_max_normal() * 2.0D0              ! Cause overflow
      x = 2.0D0 / 3.0D0                       ! Cause inexact

      flgs=ieee_flags('get','exception','',out)!which exceptions raised?

      inx   = and(rshift(flgs, fp_inexact)  , 1)  ! Decode the
      div   = and(rshift(flgs, fp_division) , 1)   ! value
      under = and(rshift(flgs, fp_underflow), 1)    ! returned in
      over  = and(rshift(flgs, fp_overflow) , 1)     ! flgs, using
      inv   = and(rshift(flgs, fp_invalid)  , 1)      ! bit-shifts

      PRINT *, "Highest priority exception is: ", out

      PRINT *, ' invalid  divide  overflo underflo inexact' !1=raised
      PRINT '(5i8)', inv,div,over,under,inx            ! 0=not raised
      i = ieee_flags('clear', 'exception', 'all', out)! Clear all
      END
```

Compile and run to detect all five exceptions with `ieee_flags`.

```
demo% f77  -silent DetAllFlg.F
demo% a.out
Highest priority exception is: invalid
 invalid  divide  overflo underflo inexact
       1       1       1       1       1
demo% ■
```

Values and `ieee_values`

The `ieee_values`(3m) file describes a collection of functions. Each function returns a special IEEE value. You can use these special IEEE entities, such as *infinity* or *minimum normal*, in a user program.

Example: A convergence test might be like this.

```
      IF ( delta .LE. r_min_normal() ) RETURN
```

The values available are listed in the table below.

Table 9-2 Functions for Using IEEE Values

IEEE Value	Double Precision	Single Precision
infinity	`d_infinity()`	`r_infinity()`
quiet NaN	`d_quiet_nan()`	`r_quiet_nan()`
signaling NaN	`d_signaling_nan()`	`r_signaling_nan()`
min normal	`d_min_normal()`	`r_min_normal()`
min subnormal	`d_min_subnormal()`	`r_min_subnormal()`
max subnormal	`d_max_subnormal()`	`r_max_subnormal()`
max normal	`d_max_normal()`	`r_max_normal()`

For the two *NaN* functions, you can assign and/or print out the values, but comparisons using either of them always yield false. To determine whether some value is a *NaN*, use the function `ir_isnan(r)` or `id_isnan(d)`.

The FORTRAN names for these functions are listed in:

- `libm_double`(3f)
- `libm_single`(3f)
- `ieee_functions`(3m)

Also read:

- `ieee_values`(3m)
- The `<f77_floatingpoint.h>` header file

9

Exception Handlers and `ieee_handler()`

Most floating-point users need to know the following about IEEE exceptions.

- What happens when an exception occurs?
- How to use `ieee_handler()` to establish a function as a signal handler
- How to write a function that can be used as a signal handler
- How to locate the exception (Where did it occur?)

To get such information, you need to generate a signal for a floating-point exception. The official UNIX name for *signal: floating-point exception* is `SIGFPE`. To generate a `SIGFPE`, establish a signal handler. The default on SPARC hardware systems is that they do *not* generate a `SIGFPE`.

Establishing a Signal Handler Function with `ieee_handler()`

To establish a function as a signal handler, pass the name of the function to `ieee_handler()`, together with the exception to watch for and the action to take. Once you establish a handler, a signal is generated whenever the particular floating-point exception occurs.

The form of invoking `ieee_handler()` is as follows.

`i = ieee_handler( action, exception, handler )`		
`action`	`character`	`"get"` or `"set"` or `"clear"`
`exception`	`character`	`"invalid"` or `"division"` or `"overflow"` or `"underflow"` or `"inexact"`
`handler`	function name	The name of the function you wrote, or `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, or `SIGFPE_ABORT`
return value	`integer`	`0=OK`

There are two general kinds of signal handler functions:

- Predefined signal handler functions
- Functions that you write yourself

The Predefined Signal Handler Functions

The predefined handlers are:

- `SIGFPE_DEFAULT` (*better to get default behavior without calling this*)
- `SIGFPE_IGNORE`
- `SIGFPE_ABORT`

Writing a Signal Handler Function

Actions taken by the function are up to you, but the *form* of the functions:

- The function must be an integer function.
- The function must have three arguments, data types as follows:

Solaris 2.x (SunOS 5.***x***)

Form— `hand5x( sig, sip, uap )`

- `hand5x` is your name for your integer function
- `sig` is an integer
- `sip` is a record which has the structure `siginfo` (see sample below)
- `uap` is not used here

Sample—Form of signal handler function, *5.x*.

```
INTEGER FUNCTION hand( sig, sip, uap ) ! Form, Handler, 5.x
INTEGER sig, location
STRUCTURE /fault_typ/
     INTEGER address
END STRUCTURE
STRUCTURE /siginfo/
     INTEGER si_signo
     INTEGER si_code
     INTEGER si_errno
     RECORD /fault_typ/ fault
END STRUCTURE
RECORD /siginfo/ sip
location = sip.fault.address
... actions you take ...
END
```

If the handler installed by `ieee_handler()` is written in FORTRAN, then the handler should not make any reference to the first argument (“`sig`” in the example above). The first argument is passed by value but expected by reference in a FORTRAN handler. The actual signal number can be referenced as `loc(sig)`.

Solaris 1.x (SunOS 4.1.***x***)

Form— `hand4x( sig, code, context )`

- `hand4x` is your name for your integer function
- `sig` is an integer
- `code` is an integer
- `context` is an array of five integers.

Sample—Form of signal handler function, *4.1.x*.

```
      INTEGER FUNCTION hand( sig, code, context ) ! Form, Handler, 4.1.x
      INTEGER sig, code, context(5)
      location = context(4)
      ... actions you take ...
      END
```

Detect an Exception by Handler (5.x/4.x)

Note – These examples show only how to call the routines to get the information. They make no attempt to teach the numerical analysis that lets you know when to call them and what to do with the information.

Example: Detect exception, by handler—*SunOS 5.x/4.x*

Solaris 1.x/ 2.x (SunOS 5.***x***/4.1.***x***)
`DetExcHan.f`

`SIGFPE` is generated whenever that floating-point exception occurs.

Then the `SIGFPE is` detected and control is passed to the `myhandler` function.

```
      EXTERNAL myhandler                              ! Main
      REAL r / 14.2 /, s / 0.0 /
      i = ieee_handler ('set', 'division', myhandler )
      t = r/s
      END

      INTEGER FUNCTION myhandler(sig,code,context)! Handler, 5.x or 4.x
*     {OK in SunOS 5.x/4.1.x since all it does is abort.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f77 -silent DetExcHan.f
demo%  a.out
abort: called
Abort (core dumped)
demo% ■
```

Locate an Exception by Handler (5.x)

Example: Locate an exception (get address) using a handler—*SunOS 5.x*.

Solaris 2.x (SunOS 5.***x***)
LocExcHan5x.F

An ***address*** is mostly for those who use such low-level debuggers as `adb`.

For how to get the ***line number***, see Section 9.6, "Debugging IEEE Exceptions," on page 195.

Caveat I/O in a handler is risky. Calling `abort()` reduces the risk.

```
#include "f77_floatingpoint.h"
     EXTERNAL hand5x                                  ! Main
     INTEGER hand5x, i, ieee_handler
     REAL r / 14.2 /,  s / 0.0 /, t
     i = ieee_handler( 'set', 'division', hand5x )
     t = r/s
     END

     INTEGER FUNCTION hand5x( sig, sip, uap)! Handler, SunOS 5.x
     INTEGER sig, location
     STRUCTURE /fault_typ/
          INTEGER address
     END STRUCTURE
     STRUCTURE /siginfo/
          INTEGER si_signo
          INTEGER si_code
          INTEGER si_errno
          RECORD /fault_typ/ fault
     END STRUCTURE
     RECORD /siginfo/ sip
     location = sip.fault.address
     WRITE (*,10)  location
10   FORMAT('Exception at hex address ', Z8 )
     CALL abort()  !  Just to reduce risk
     END
demo% ■
```

Solaris 2.x (SunOS 5.x)
Compile/Load/Run

The actual address varies with installation and architecture.

```
demo% f77 -silent LocExcHan5x.F
demo%  a.out
Exception at hex address    10DC4
abort: called
Abort (core dumped)
demo% ■
```

Note – How to call, not when to call or what to do with the information.

Locate an Exception by Handler (4.1.x)

Example: Locate an exception (get address) using a handler—*SunOS 4.x*.

Solaris 1.x (SunOS 4.1.***x***)
LocExcHan4x.f
SunOS 4.x

Caveat I/O in a handler is risky. Calling `abort()` reduces the risk.

```
      EXTERNAL hand4x                              ! Main
      INTEGER hand4x, i, ieee_handler
      REAL r /14.2/,  s /0.0/, t
      i = ieee_handler('set', 'division', hand4x)
      t = r / s
      END

      INTEGER FUNCTION hand4x(sig,code,context) ! Handler , SunOS4.x
      INTEGER sig, code, context(5)
      WRITE( *, '("Exception at pc", I5 )' ) context(4)
      CALL abort() !  Just to reduce risk
      RETURN
      END
```

Solaris 1.x (SunOS 4.1.***x***)
Compile/Load/Run

The actual address varies with installation and architecture.

```
demo% f77 -silent LocExcHan4x.f
demo%  a.out
Exception at pc 8980
abort: called
Abort
demo% ■
```

Note – How to call, not when to call or what to do with the information.

Detect All Exceptions by Handler (Solaris 2.x, SunOS 5.x)

Example: Detect & locate all exceptions, with a signal handler—*SunOS 5.x*.

Solaris 2.x (SunOS 5.x)
`DetAllHan5x.F`
Main →

Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

.

```
#include "f77_floatingpoint.h"
        DOUBLE PRECISION x, y, d_max_normal, d_min_normal, z/0.0d0/
        EXTERNAL continue5x
        INTEGER continue5x
       ieeer = ieee_handler('set', 'all', continue5x)    ! Establish handler
        IF (ieeer.ne.0) PRINT *,'cannot establish handler: continue5x'
      ieeer = ieee_handler('set', 'inexact', SIGFPE_IGNORE) ! Ignore inexact

        WRITE(*,"(/'0/0:')")
        x = 0.0d0 / z                                      ! Invalid
        WRITE(*,"(/'3.14159/0.0 Trapped:')")
        x = 3.14159d0 / z                                 ! Div by 0, trapped

        WRITE(*,"(/'max_normal**2:')")
        y = d_max_normal()
        x = y * y                                          ! Overflow
        WRITE(*,"(/'min_normal**2:')")
        y = d_min_normal()
        x = y * y                                          ! Underflow

        ieeer = ieee_handler('set', 'inexact', continue5x)! Trap inexact
        IF (ieeer.ne.0) PRINT *,"Can't set inexact, handler continue5x"
        WRITE(*,"(/'2.0/3.0:')")
        x = 2.0d0 / 3.0d0                                   ! Inexact

       ieeer = ieee_handler('clear','division',SIGFPE_DEFAULT)! Set div dflt
        IF (ieeer.ne.0) PRINT *, 'cannot clear division handler'
        WRITE(*,"(/'3.14159/0.0 Untrapped:')")
        x = 3.14159d0 / z                               ! Div by 0, untrapped
        WRITE(*,"(' 3.14159/0.0 = ', F12.8/)") x
        END
... continued ...
```

Note – How to call, not when to call or what to do with the information.

Example: Detect & locate all exceptions, with handler—*SunOS 5.x* (*continued*).

Solaris 2.x (SunOS 5.***x***)
DetAllHan5x.F
(*continued*)
Handler ⟶

< Uses `sysmachsig.h` values > which is schlepped in by `f77_floatingpoint.h`.

These codes may be different on Solaris x86. Check the `</usr/include/sys/machsig.h>` file.

```
      INTEGER FUNCTION continue5x(sig, sip, uap) ! Handler-5.x
      INTEGER sig, code, location
      CHARACTER*9 label
      STRUCTURE /fault_typ/
           INTEGER address
      END STRUCTURE
      STRUCTURE /siginfo/        ! Translate siginfo_t in <sys/siginfo.h>
           INTEGER si_signo
           INTEGER si_code
           INTEGER si_errno
           RECORD /fault_typ/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      code = sip.si_code    ! Which exception raised SIGFPE?
      IF (code .eq. 3) label = 'division' ! These
      IF (code .eq. 4) label = 'overflow'  ! 5 codes
      IF (code .eq. 5) label = 'underflow'  ! are defined
      IF (code .eq. 6) label = 'inexact'     ! in
      IF (code .eq. 7) label = 'invalid'      ! <sys/machsig.h>
      location = sip.fault.address
      WRITE(*,10) label, code, location   ! I/O in handler is risky
10    FORMAT(A10,' exception, sigfpe code',I2,',at address
',Z8)
      END
```

Solaris 2.x (SunOS 5.x)

Compile/Load/Run

The addresses vary, depending on installation and architecture. The addresses are mostly for those who use such low-level debuggers as adb. See page 196 for how to get the source line number.

```
demo$ f77 -silent DetAllHan5x.F
demo% a.out
0/0:
 invalid   exception, sigfpe code 7, at address    11144

3.14159/0.0 Trapped:
 division  exception, sigfpe code 3, at address    11190
 max_normal**2:
 overflow  exception, sigfpe code 4, at address    111DC
min_normal**2:
 underflow exception, sigfpe code 5, at address    11228
2.0/3.0:
 inexact   exception, sigfpe code 6, at address    1130C
3.14159/0.0 Untrapped:
 3.14159/0.0 = Infinity
 ... retrospective messages about exceptions ...
demo% ■
```

Example: Detect & locate all exceptions, with a signal handler—*SunOS 4.x*.

Solaris 1.x (SunOS 4.1.***x***)
`DetAllHan4x.F`
Main⟶
Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

```
#include "f77_floatingpoint.h"
        DOUBLE PRECISION x, y, d_max_normal, d_min_normal, z/0.0d0/
        EXTERNAL continue4x
        INTEGER continue4x
       ieeer = ieee_handler('set', 'all', continue4x)   ! Establish handler
       IF (ieeer.ne.0) PRINT *,'cannot establish handler: continue4x'
      ieeer = ieee_handler('set','inexact',SIGFPE_IGNORE)! Ignore inexact

        WRITE(*,"(/'0/0:')")
        x = 0.0d0 / z                                      ! Invalid
        WRITE(*,"(/'3.14159/0.0:')")
       x = 3.14159d0 / z                                 ! Div by 0, trapped

        WRITE(*,"(/'max_normal**2:')")
        y = d_max_normal()
        x = y * y                                          ! Overflow
        WRITE(*,"(/'min_normal**2:')")
        y = d_min_normal()
        x = y * y                                          ! Underflow

        ieeer = ieee_handler('set', 'inexact', continue4x)! Trap inexact
        IF (ieeer.ne.0) PRINT *,'Cannnot establish handler: inexact'
        WRITE(*,"(/'2.0/3.0:')")
        x = 2.0d0 / 3.0d0                                  ! Inexact

      ieeer=ieee_handler('clear','division',SIGFPE_DEFAULT)! Div default
        IF (ieeer.ne.0) PRINT *, 'could not clear division handler'
        WRITE(*,"(/'3.14159/0.0:')")
        x = 3.14159d0 / z                                 ! Div by 0, untrapped
        WRITE(*,"(' 3.14159/0.0 = ', F12.8/)") x
        END

        INTEGER FUNCTION continue4x(sig,code,sigcontext) ! Handler- 4.x
        INTEGER code, sig, sigcontext(5)
        CHARACTER label*16
        IF (loc(code) .eq. 208) label = 'invalid'
        IF (loc(code) .eq. 200) label = 'division by zero'
        IF (loc(code) .eq. 212) label = 'overflow'
        IF (loc(code) .eq. 204) label = 'underflow'
        IF (loc(code) .eq. 196) label = 'inexact'
       WRITE (*,1) loc(code), label, sigcontext(4) ! I/O in handler is risky
 1      FORMAT(' ieee exception code',I4, ',', A17, ',', ' at pc',I6)
        END
```

The next page has ⟶ details on `inexact`.

Handler⟶

Note special codes

Example: Detect & locate all exceptions, with handler—*SunOS 4.x* (*continued*)

Solaris 1.x (SunOS 4.1.***x***) Compile/Load/Run

The addresses vary, depending on installation and architecture. The addresses are mostly for those who use such low-level debuggers as adb.

See page 196 for how to get the source line number.

```
demo$ f77 -silent DetAllHan4x.F
demo% a.out

0/0:
 ieee exception code 208, invalid         , at pc  9176

3.14159/0.0 Trapped:
 ieee exception code 200, division by zero, at pc  9252

max_normal**2:
 ieee exception code 212, overflow        , at pc  9328

min_normal**2:
 ieee exception code 204, underflow       , at pc  9404

2.0/3.0:
 ieee exception code 196, inexact         , at pc  9632

3.14159/0.0 Untrapped:
 3.14159/0.0 = Infinity

 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Division by Zero;
 Note: IEEE Infinities were written to ASCII strings or output files;
    see econvert(3).
 Note: Following IEEE floating-point traps enabled;
   see ieee_handler(3M):
 Inexact;  Underflow;  Overflow;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
demo% ■
```

In the above example, after the execution of `x=2.0d0/3.0d0`, `x` contains:

inexact

	SunOS 5.*x*	**SunOS 4.1.3 and Later**	**SunOS Before 4.1.3**
Untrapped `inexact`	0.666…	0.666…	0.666…
Trapped `inexact`	*garbage*	*garbage*	0.666…

The value is garbage because it is unpredictable; it depends on various actions that happen immediately before the exception.

9

Retrospective

The `ieee_retrospective` function queries the floating-point status registers to find out which exceptions have accrued. If any exception has a raised accrued exception flag, a message is printed to standard error to inform the programmer which exceptions were raised but not cleared. For FORTRAN, this function is called automatically just before normal termination

The message typically looks like this (format varies with each release):

```
NOTE: The following IEEE floating-point arithmetic
exceptions occurred and were never
cleared: Inexact; Division by Zero; Underflow;
Overflow; Invalid Operand;
Sun's implementation of IEEE arithmetic is discussed
in the Numerical Computation Guide
```

Nonstandard Arithmetic

Another useful math library function is *nonstandard arithmetic*. The IEEE standard for arithmetic specifies a way of handling underflowed results gradually, by dynamically adjusting the radix point of the significand. Recall that in IEEE floating-point format, the radix point occurs before the significand, and there is an implicit leading bit of 1. Gradual underflow allows the implicit leading bit to be cleared to 0, and to shift the radix point into the significand, when the result of a floating-point computation would otherwise underflow. This is not accomplished in hardware on a SPARC processor, but in software. If your program happens to generate many underflows (perhaps a sign of a problem with your algorithm?), and you run on a SPARC processor, you may experience a performance loss.

To turn off gradual underflow, compile with `-fnonstd`, or insert this.

```
      CALL nonstandard_arithmetic()
```

To turn on gradual underflow (after you have turned it off), insert this.

```
      CALL standard_arithmetic()
```

- The `standard_arithmetic()` subroutine corresponds exactly to an earlier version named `gradual_underflow()`.
- The `nonstandard_arithmetic()` subroutine corresponds exactly to an earlier version named `abrupt_underflow()`.

Messages about Floating-point Exceptions

For FORTRAN, the current default is to display a list of accrued floating-point exceptions at the end of execution. In general, you will get a message if any one of the invalid, division-by-zero, or overflow exceptions occur. Since most real programs raise inexact exceptions, you will get a message if exceptions other than inexact exceptions occur. If only inexact, then no message.

You can turn off any or all of these messages with `ieee_flags()` by clearing exception status flags. If this is done at all, it is usually done at the end of your program. You can gain complete control with `ieee_handler()`.

In your own exception handler routine you can:

- Specify actions
- Turn off messages with `ieee_flags()` by clearing exception status flags

---

Note – Clearing all messages is not recommended. If you need to turn off these messages, it is recommended that you record *invalid*, *divide by zero*, and *overflow* someplace.

9.6 Debugging IEEE Exceptions

You may want to debug programs that have worrisome messages like this.

```
NOTE: the following IEEE floating-point arithmetic
exceptions occurred and were never
cleared: Inexact; Division by Zero; Underflow;
Overflow; Invalid Operand;
Sun's implementation of IEEE arithmetic is discussed
in the Numerical Computation Guide
```

To locate the *line number* where the exception occurred, do the following:

- Establish a signal handler so that a `SIGFPE` gets generated.
- After you invoke `dbx`, enter the "`catch FPE`" command.

Locating such a line number is shown in the following example, but see page 184 for details about exception handlers.

Example: Locate the *line number* of an exception, dbx / handler—*SunOS 5.x/4.x*.

Solaris 1.x/2.x (SunOS 5.x/4.x)
LocExcDbx.f
SunOS 5.x/4.x

You can find the source code line where a floating-point exception occurred by using the ieee_handler routine with either dbx or debugger.

Note the "catch FPE" dbx command. →

```
demo$ cat LocExcDbx.f
      INTEGER myhandler                                    ! Main
      EXTERNAL myhandler
      REAL r /14.2/, s /0.0/
      ieeer = ieee_handler('set', 'common', myhandler)
      PRINT *, r/s
      END

      INTEGER FUNCTION myhandler( sig, code, context ) ! Handler
!     {OK in Solaris 2.x/1.x, since all it does is abort.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo$ f77 -g -silent LocExcDbx.f
demo$ dbx a.out
Reading symbolic information …
(dbx) catch FPE
(dbx) run
Running: a.out
signal FPE (floating point exception)
           in MAIN at line 5 in file "LocExcDbx.f"
    5           PRINT *, r/s
(dbx) quit
demo$ ■
```

9.7 Guidelines

To sum up, SPARC arithmetic is a state-of-the art implementation of IEEE arithmetic, optimized for the most common cases.

- More problems can safely be solved in single precision, due to the clever design of IEEE arithmetic.
- To get the benefits of IEEE math for most applications, if your program gets one of the common exceptions, then you probably want to continue with a sensible result. That is, you do *not* want to use ieee_handler to *abort* on the common exceptions.
- If your system time is very large, over 50% of runtime, *check into* modifying your code, or using nonstandard_arithmetic.

9.8 Miscellaneous Examples

A miscellaneous collection of more or less realistic examples is provided here, as a possible additional aid.

Kinds of Problems

The problems in this chapter usually involve arithmetic operations with a result of invalid, division by zero, overflow, underflow, or inexact.

For instance, *Underflow* — In *old* arithmetic, that is, prior to IEEE, if you multiply two very small numbers on a computer, you get zero. Most mainframes and minicomputers behave that way. In IEEE arithmetic, there is *gradual underflow*; this expands the dynamic range of computations.

For example, consider a machine with `1.0E-38` as the machine *epsilon*, the smallest representable value on the machine. Multiply two small numbers.

```
a = 1.0E-30
b = 1.0E-15
x = a * b
```

In old arithmetic you get `0.0`, but with IEEE arithmetic (and the same word length) you get `1.40130E-45`. With old arithmetic, if a result is near zero, it becomes zero. This can cause problems, especially when subtracting two numbers — because this is a principal way accuracy is lost.

You can also detect that the answer is inexact. The `inexact` exception is common, and means the calculated result cannot be represented exactly, at least not in the precision being used, but it is as good as can be delivered.

Underflow tells us, as we can tell in this case, that we got an answer smaller than the machine naturally represents. This is accomplished by stealing some bits from the mantissa and shifting them over to the exponent. The result is less precise, in some sense, but more so in another. The deep implications are beyond this discussion. The interested reader may wish to consult *Computer,* January 1980, Volume 13, Number 1, particularly I. Coonen's article, "Underflow and the Denormalized Numbers."

Roundoff — Most scientific programs have sections of code that are sensitive to roundoff, often in an equation solution or matrix factorization. So be concerned about numerical accuracy — if your computer doesn't do a good job, your results will be tainted, and there is often *no way to know* that this has happened.

Simple Underflow

Some applications actually do a lot of work very near zero. This is common in algorithms which are computing residuals, or differential corrections. For maximum numerically safe performance, perform the key computations in extended precision. If the application is a single-precision application, this is easy, as we can perform key computations in double precision.

Example: A simple dot product computation.

```
sum = 0
DO i = 1, n
     sum = sum + a(i) * b(i)
END DO
```

If `a(i)` and `b(i)` are small, many underflows will occur. By forcing the computation to double precision, you compute the dot product with greater accuracy, and not suffer underflows.

```
REAL*8 sum
DO i = 1, n
     sum = sum + dble(a(i)) * dble(b(i))
END DO
result = sum
```

It may be advisable to have both versions, and to switch to the double precision version only when required.

You can force a SPARC processor to behave like an older computer with respect to underflow. Add the following to your FORTRAN main program.

```
CALL nonstandard_arithmetic()
```

But be aware that you are giving up the numerical safety belt that is the operating system default. You can get your answers faster, and you won't be any less safe than, say, a VAX — but use at your own risk.

Use Wrong Answer

You might wonder why continue if the answer is clearly *wrong*. The general idea is that IEEE arithmetic allows you to make distinctions about what kind of *wrong*, such as `NaN` or `Inf`. Then decisions can be made based on such distinctions.

For an example, consider a circuit simulation. The only variable of interest (for the sake of argument) from a particular 50 line computation is the voltage. Further assume that the only values which are possible are +5v, 0, -5v.

It is possible to carefully arrange each part of the calculation to coerce each subresult to the correct range.

```
      4.0  < computed <  Inf →  5 volts
     -4.0  ≤ computed ≤  4.0 →  0 volts
     -Inf  < computed ≤ -4.0 → -5 volts
```

Furthermore, since `Inf` is not an allowed value, you need special logic to ensure that big numbers are not multiplied.

IEEE arithmetic allows the logic to be much simpler, as the computation can be written in the obvious fashion, and only the final result need be coerced to the correct value, since ±`Inf` can occur, and can be easily tested.

Furthermore the special case of 0/0 can be detected and dealt with as you wish. The result is easier to read, and faster executing (since you don't do unneeded comparisons).

Excessive Underflow

If two very small numbers are multiplied, the result underflows.

- For some SPARC platforms, the hardware, being designed for the typical case, does *not* produce a result; instead software is employed to compute the correct IEEE complying result. As one might guess, this is much slower. In the majority of applications, this is invisible. When it is not, the symptom is that the system time component of your runtime (which can be determined by running your application with the `time` command) is much too large.
- For other SPARC platforms, the hardware *does* produce the result. This is much faster.

The following examples have varying differences depending on the platform.

Example: Excessive underflow.

DotProd.f

```
      PROGRAM dotprod
      INTEGER maxn
      PARAMETER (maxn=10000)
      REAL a(maxn), b(maxn), eps /1.0e-37/, sum
      DO i = 1, maxn
           a(i) = 1.0e-30
           b(i) = 1.0e-15
      END DO
      sum = 0.
      DO i = 1, maxn
           sum = sum + a(i)*b(i)
      END DO
      END
```

After compiling and running `dotprod`, the results of the `time` command are:

```
        2.3 real         0.1 user         1.8 sys
```

The real computation took about 0.1 seconds, but the software fix took 2 seconds. In a real application the difference can be *hours*. This is not desirable.

Solution 1: Change All of the Program

If you rewrite with double precision variables you get vast improvement.

```
        0.2 real         0.0 user         0.1 sys
```

It may not be desirable to promote an entire program to double precision (though this is what is traditionally done to make up for the fact that old style arithmetic is less accurate).

Solution 2: Change One Double-Precision Variable

Declare only `sum` to be double precision, and change only the summation line of code as follows:

```
      sum = sum + a(i)*dble(b(i))
```

This minimizes the software underflow problem.

```
        0.3 real         0.1 user         0.0 sys
```

Note that in a real application, you should put the variable `sum` in double precision and coerce it to single precision only on output. This is not a performance issue, but a numeric one. Of course it may not be easy to tell which variables in a huge program need to be promoted. The effort is worthwhile, not only because of the performance (which, as you will learn, can be achieved in other ways), but because the numerics are enhanced as well.

Solution 3: Nonstandard Arithmetic

There is a quick and dirty solution, which is:

```
      CALL nonstandard_arithmetic()
```

This tells the hardware to act like an old-style computer, and when underflow occurs just flush to zero. This results in a runtime as follows.

```
        0.5 real         0.0 user         0.1 sys
```

Note that this time is about the same as promoting one variable to double. The difference is that now the computed result is 0. This is bad because if this dot product is really the final result, there is probably nothing wrong with this solution. If, however, this result feeds into more elaborate computations, you have thrown away some information. This may be important. If the algorithm is stable, the input well conditioned, and the implementation careful, it won't matter. If there is anything else shaky, this may push it over.

Solution 4: The `-r8` *Option*

Another quick fix is the `-r8` option. This is safe, but just a bit costly. It informs the compiler to interpret `REAL` as `DOUBLE PRECISION`. The `-r8` solution may be preferred when the code was developed on a CRAY, CDC, or other 64-bit machine. Note that in many cases `-r8` will suffice to produce correct results (thanks to the miracles of modern arithmetic) and will be faster.

If you recompile `DotProd.f` with `-r8`, you get the following from `time`.

```
        0.8 real         0.0 user         0.1 sys
```

If you wish to dig, it will be helpful to read the section on `ieee_handler` and to employ it to track down the affected lines.

`-r8` *with Migrating*

Users migrating from chips like 68881 or 80387 processors may wonder why `-r8` is necessary. The code worked fine (full speed) on their last machine. The reason is that these numeric processors provide internal registers which are 80-bits wide.

The good news about an 80-bit FPU:

- When everything fits in the 80-bit registers, the results are a little better.

The bad news about an 80-bit FPU:

- An 80-bit FPU is typically slower and/or more expensive than either a 32-bit or a 64-bit FPU.
- Since some intermediate results are computed with 80-bit precision, and others with only 32-bit or 64-bit precision, answers depend on exactly how the code is written, what optimization level is selected, compiler version, and other factors not under user control. Results tend to vary, resulting in harder to validate software, and so forth.

At this point, every SPARC processor performs arithmetic with 32 or 64-bit precision as coded by the user.

If you are porting codes that were developed on old arithmetic machines, it is probably preferable to *stop* on overflows, division by zero, and so forth. A solution is to use the `ieee_handler`, as in the examples.

`-dalign`

If `–r8` is combined with `–dalign`, the program will run slower than without the `-r8` option. This is likely to happen if the key computational loops are very heavily exercised *and* involve mixed precision (double + single).

`-r8` *with Double Precision*

If `–r8` is used and the key computational loops are very heavily exercised *and* involve double precision, then on SPARC platforms, the program will run slower than without the `-r8` option. The double precision is converted to quadruple precision, which is slower.

Porting from Other FORTRANs 10

This chapter is organized into the following sections.

General Hints	*page 203*
Time Functions	*page 204*
Formats	*page 207*
Carriage-Control	*page 207*
File Equates	*page 208*
Data Representation	*page 208*
Hollerith	*page 209*
Porting Steps	*page 211*

This chapter introduces porting programs from other dialects of FORTRAN. If you have VMS FORTRAN programs, most compile almost exactly as is, but if they don't, read about VMS extensions in the *FORTRAN Reference Manual.*

10.1 General Hints

Keep these conventions in mind when transporting from another machine.

- Your source file name must have a `.f`, `.F`, or `.for` extension.
- If entering programs manually (instead of reading them from tape), start lines with a tab or space so code begins after column five, except for comments and labels.

10.2 Time Functions

When porting programs from a different FORTRAN system, check the code to make sure that time functions used in the programs operate like those in this FORTRAN. If they do not, change the program to use equivalent functions.

These time functions (found on some other machines) are not directly supported, but you can write subroutines to duplicate their function.

- Time-of-day in `10h` format
- Date in `A10` format
- Milliseconds of job CPU time
- Julian date in ASCII

For example, to find the current Julian date, call `TIME()` to get the number of seconds since January 1, 1970, convert the result to days (divide by 86,400), and add 2,440,587 (the Julian date of December 31, 1969).

Several time functions are supported in the `f77` extensions to standard FORTRAN and are described in the following two tables.

Table 10-1 Time Functions Available to FORTRAN

Name	Function	Man Page
time	Returns the number of seconds elapsed since 1 January, 1970	time(3f)
fdate	Returns the current time and date as a character string	fdate(3f)
idate	Returns the current month, day, and year in an integer array	idate(3f)
itime	Returns the current hour, minute, and second in an integer array	itime(3f)
ctime	Converts time returned by `time` function to character string	ctime(3f)
ltime	Converts time returned by `time` function to local time	ltime(3f)
gmtime	Converts time returned by `time` function to Greenwich time	gmtime(3f)
etime	*Single Processor:* Returns elapsed user and system time for program execution *Multiple Processors:* Returns wall clock time	etime(3f)
dtime	Returns elapsed user and system time since last call to `dtime`	dtime(3f)

These next routines provide compatibility with VMS FORTRAN system routines. To use these routines you must include the `-lV77` option on the `f77` command line, in which case you will also get the VMS versions of `idate` and `time`, instead of the standard versions.

Example. Using the `-lV77` option.

```
demo$ f77 myprog.f -lV77
```

Table 10-2 Summary: VMS FORTRAN System Routines

Name	Definition	Calling Sequence	Argument Type	Returned Type
date ♦	Date as dd-mmm-yy	`call date( c )`	`character*9`	n/a
idate ♦	Date as d, m, y	`call idate( d, m, y )`	`integer`	n/a
secnds♦	Time of day or elapsed time	`t = secnds( u )`	`real`	`real`
time ♦	Current time as hhmmss	`call time( t )`	`character*8`	n/a
ran ♦	Random number	`r = ran( s )`	`integer*4`	`real`
mvbits♦	Move bit field	`call mvbits(src,i1,n,des,i2)`	`integer`	n/a

The error condition subroutine `errsns` is *not* provided because it is totally specific to the VMS operating system. The terminate program subroutine `exit` was already provided by the operating system.

Sample implementation of time functions that might appear on other systems:

```
      subroutine startclock
      common / myclock / mytime
      integer mytime
      integer time
      mytime = time()
      return
      end
      function wallclock
      integer wallclock
      common / myclock / mytime
      integer mytime
      integer time
      integer newtime
      newtime = time()
      wallclock = newtime - mytime
      mytime = newtime
      return
      end
      integer wallclock, elapsed
      character*24 greeting
      real dtime
      real timediff, timearray(2)
c     print a heading
      call fdate( greeting )
      write( 6, 10 ) greeting
10    format('1hi, it''s ', a24 /)
c     see how long an 'ls' takes, in seconds
      call startclock
      call system( 'ls' )
      elapsed = wallclock()
      write( 6, 20 ) elapsed
20    format(//,'elapsed time ', i4, ' seconds'///)
c     now test the cpu time for some trivial computing
      timediff = dtime( timearray )
      q = 0.01
      do 30 i = 1, 1000
           q = atan( q )
30    continue
      timediff = dtime( timearray )
      write( 6, 40 ) timediff
40    format(//,'computing atan(q) 1000 times',
&           /   'took ', f6.3,' seconds.'/)
      end
```

10.3 Formats

Some f77 format features may be different from the formats provided in other versions of FORTRAN. Even when the formats used in other FORTRAN implementations are different, with a little care programs are still often transportable to f77. Here are some format specifiers that f77 treats differently than some other implementations:

A

Used with character type data elements. In FORTRAN 66, this specifier worked with any variable type. f77 supports the older usage, up to four characters to a word

$

Suppresses newline character output

R

Sets an arbitrary radix for the I formats that follow in the descriptor

SU

Select unsigned output for following I formats. For example, you can convert output to either hexadecimal or octal with the following formats, instead of using the Z or O edit descriptors.

```
10   FORMAT( SU, 16R, I4 )
20   FORMAT( SU, 8R, I4 )
```

10.4 Carriage-Control

FORTRAN carriage-control grew out of the capabilities of the equipment used when FORTRAN was originally being developed. For similar historical reasons, an operating system, derived from the UNIX operating system, doesn't have FORTRAN carriage-control, but you can simulate it in two ways.

- For simple jobs, use OPEN(N, FORM='PRINT'). You then get single or double spacing, formfeed, and stripping off of column one. It is legal to reopen unit 6 to change the form parameter to PRINT, for example

```
     OPEN( 6, FORM='PRINT')
```

- Use the `asa` filter to transform FORTRAN carriage-control conventions into the UNIX carriage-control format (see the `asa` (1) man page) before printing files using `lpr`.

10.5 File Equates

Early versions of FORTRAN did not use named files, and file equates provided some ability to open files by name. You can use pipes and I/O redirection, as well as hard or soft links, in place of file equates in transported programs.

Example: Redirect `stdin` from `redir.data`. This example uses `csh`(1).

```
demo% cat redir.data      ← The data file
 9 9.9

demo% cat redir.f         ← The source file
      read(*,*) i, z
      print *, i, z
      stop
      end

demo% f77 redir.f         ← The compile
redir.f:
 MAIN:
demo% a.out < redir.data ← Run with redirection
  9 9.90000
demo% ■
```

See Chapter 4, “File System and FORTRAN I/O” for more on piping and redirection.

10.6 Data Representation

Read the appendix “Data Representations” in the *FORTRAN Reference Manual* for the exact representation of different kinds of data in FORTRAN. This section points out information necessary for transporting FORTRAN programs. Remember the following:

- Because we adhere to the IEEE 754 standard for floating-point, the first four bytes in a `REAL*8` are not the same as in a `REAL*4`.

- The default sizes for reals, integers, and logicals are the same (according to the FORTRAN standard) except when the `-i2` flag is used, which shrinks integers and logicals to two bytes but leaves reals as four bytes.
- Character variables can be freely mixed and equivalenced with variables of other types, but be careful of potential alignment problems.
- SPARCsystem floating-point arithmetic does raise exceptions on overflow or divide-by-zero, but does not signal SIGFPE by default. It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. Read the appendix "*Data Representations*" in the *FORTRAN Reference Manual*.
- The extreme finite, normalized values may be determined, see `libm_single(3f)` and `libm_double(3f)`. The indeterminate forms can be written and read using formatted and list-directed I/O statements.

10.7 Hollerith

This section is useful for porting older programs—not for writing or heavily modifying a program. It is recommended that you use character variables for the purpose covered in this section. You can initialize variables with the older FORTRAN Hollerith (*n*`H`) feature, but remember that this is not standard.

Table 10-3 Maximum Characters per Data Type

Data Type	Max Characters Per Datum
`BYTE*1`	1
`COMPLEX`	8 (or 16 with `-r8`)
`COMPLEX*16`	16
`COMPLEX*32`	32
`DOUBLE COMPLEX`	16 (or 32 with `-r8`)
`DOUBLE PRECISION`	8
`INTEGER`	4 (or 2 with `-i2`, or 8 with `-r8`)
`INTEGER*2`	2
`INTEGER*4`	4
`LOGICAL`	4 (or 2 with `-i2`, or 8 with `-r8`)
`LOGICAL*1`	1
`REAL`	4 (or 8 with `-r8`)
`REAL*4`	4
`REAL*8`	8
`REAL*16`	16

Example: Initialize variables with Hollerith.

```
      double complex x(2)
      data x /16HHello there, sai, 16Hlor, new in town/
      write( 6, '(4A8, "?")' ) x
      end
```

If you pass Hollerith constants as arguments, or if you use them in expressions or comparisons, they are interpreted as character-type expressions.

If you must, you can initialize a data item of a compatible type with a Hollerith, and then pass it around.

Example:

```
        integer function doyouloveme()
        double precision fortran, beloved
        integer yes, no
        data yes, no / 3hyes, 2hno /
        data fortran/ 7hFORTRAN/
10      format( "Whom do you love? ", $ )
        write( 6, 10 )
        read ( 5, 20 ) beloved
20      format( a8 )
        doyouloveme = no
        if ( beloved .eq. fortran ) doyouloveme = yes
        return
        end
```

```
        program trouble
        integer yes, no
        integer doyouloveme
        data yes, no / 3hyes, 2hno /

        if ( doyouloveme() .eq. yes ) then
            print *, 'You are sick'
        else
            print *, 'See if I ever speak to you again'
        endif
        end
```

All these things produce warning messages from the compiler.

10.8 Porting Steps

This outline of steps does not contain *all* that you need to know about porting. It leads into performance issues, which is the topic of the next chapter. It is designed for someone who is stuck with doing a large job in a short time — and who probably does not code in FORTRAN on a daily basis.

Typical Case

How To Port FORTRAN for Fun and Profit

Sample situation:

- The user-provided code is of modest size (10K lines).
- All the subroutines are contained in one file.
- A simple "`f77 -O prog.f`" breaks.

What to do?

1. For your own protection and peace of mind, first save a complete set of original files (including any `READ.ME` files, `.COM` files, etc.).

2. Make a new directory (say `src`), and copy your files to it, and go there.

```
demo$ mkdir src
demo$ cd src
```

3. Split the multi-subroutine, single file of source into many files, one subroutine per file.

```
demo$ fsplit ../prog.f
```

   This may produce a whole lot of files. `fsplit` may not always work, so don't throw `prog.f` away.

4. Create a `makefile`.

```
FFLAGS = -fast $(FLAGS)
OBJ = subs.o main.o

example: $(OBJ)
     f77 $(FFLAGS) $(OBJ) -pg -o example \
          -Bstatic -lm
```

It is important to remember that in general the makefile defaults may be inappropriate for your particular code: The default is to use `-cg89`. All current models use `-cg89` hardware.

Performance issues start their insidious creep about here.

5. If the `-pg` option were placed on the `ld` line, that would result in a profile that does not include the columns "`#calls time/call`" (because the individual routines were not compiled with `-pg`). That is why the `-pg` is on the compile line.

6. Compile all the source files with one make file command.

```
demo$ make
```

Since we selected `–fast` as the default compilation flag in the makefile, we have implicitly asked for (among other options) the `–O3` level of optimization.

7. Execute the code.

```
demo$ example
```

8. Check the answers; make sure they are correct.

9. Run `gprof`.

```
demo$ gprof example > profile
```

Examine the profile reports of `gprof`, using `more` or your favorite editor.

The report comes in two parts, a flat profile and a *call graph* report. The flat report comes second, and can be quickly found by searching for the "flat" string.

You may want to recompile the most expensive routines (those coming first in the flat report) with "`-fast -O4`". This can be accomplished either by compiling them by hand, or by editing the makefile. A simplistic makefile rewrite looks like this.

```
FFLAGS = -fast $(FLAGS)
OBJ = subs.o main.o

example: $(OBJ)
     f77 $(FFLAGS) $(OBJ) -pg -o example \
     -Bstatic -lm

expensive_routinec.o: expensive_routine.f
     f77 -fast -O4 -c expensive_routine
```

If answers are OK and the timing information is fast enough (that is. within say 20% of your target), you are done. If it is not fast enough, someone must do some code tuning.

What To Do When Things Go Wrong

If the Answers Are Close, but Not Right On

- Pay attention to size and the engineering units. Numbers very close to zero can appear to be different, but the difference is not significant. For example 1.9999999e-30 ≈ -9.9992112e-33, especially if this number is the difference between two large numbers, such as the distance across the continent in feet, as calculated on two different computers. VAX math is not as good as IEEE math, and even different IEEE processors may differ. This is especially true if it involves lots of trig functions. These are much more complicated than one might think, and the standard rigorously defines only the basic arithmetic functions, so there can be subtle differences, even between IEEE machines.
- Try running with `call nonstandard_arithmetic`. This can also improve performance considerably. This will make your Sun behave more like a VAX. If you have a VAX (or some other computer) handy, run it there also. It is quite common for many numerical applications to produce slightly different results on each floating-point implementation.

- Check for `NaN`, `+Inf`, and other signs of weirdness. See Section 9.5, “IEEE Routines” or the `man` page `ieee_handler`(3m) for instructions on how to trap the various exceptions. On most machines these goodies will simply kill the run.
- Two numbers can differ by 6 x 10^{29} but have the same floating-point form. Example: Different numbers, same representation.

```
      real*4 x,y
      x=99999990e+29
      y=99999996e+29
      write (*,10), x, x
 10   format('99,999,990 x 10^29 = ', e14.8, ' = ', z8)
      write(*,20) y, y
 20   format('99,999,996 x 10^29 = ', e14.8, ' = ', z8)
      end
```

The output is as follows.

```
99,999,990 x 10^29 = 0.99999993E+37 = 7cf0bdc1
99,999,996 x 10^29 = 0.99999993E+37 = 7cf0bdc1
```

In this example the difference is 6 x 10^{29}. The reason for this (indistinguishable) wide gap is that in IEEE single precision you're only guaranteed 6 decimal digits (for any one decimal-to-binary conversion.) You might get 7 or 8 digits converted correctly but it depends on the number.

If the Program Blows Up without Warning

If the program blows up without warning and it runs different lengths of time between failures, then:

- Turn off the optimizer. If it then works, turn it back on for only the top routines.
- Understand that optimizers must make assumptions about the program. If the user has done some non-standard things, this can cause problems (like the SAVE statement). Almost no optimizer handles *all* programs at *all* levels of optimization.

Before calling for help, make sure you have the current software (such as FORTRAN 3.0.1 and Solaris 2.x or Solaris 1.x) and you are either under warranty or have a software support contract.

Profiling 11

This chapter is organized into the following sections.

Summary	*page 215*
The time Command	*page 216*
The gprof Command	*page 218*
The tcov Command	*page 221*
I/O Profiling	*page 223*
Missing Profile Libraries	*page 224*

11.1 Summary

This chapter describes measuring the resources used by programs. For another way to profile, read the manual *Performance Tuning an Application*, in the SPARCworks set.

Example: Main for profiling.

`p1.f`
This program is used in several examples. It is a revised version of the one in Chapter 8, “Debugging and it calls `mkidentity` 100,000 times.

```
      program silly
      paramater (n=2)
      real twobytwo(2,2) / 4 *-1 /
      do i = 1, 100000
           call mkidentity( twobytwo, n )
      end do
      print *, determinant(twobytwo)
      end
```

Example: Subroutine for profiling.

p2.f

```
      subroutine mkidentity(matrix,dim)
      real matrix(dim,dim)
      integer dim
      do 90 m = 1, dim
           do 20 n = 1, dim
                if(m.eq.n) then
                        matrix(m,n) = 1.
                else
                        matrix(m,n) = 0.
                endif
20         continue
90    continue
      return
      end
```

Example: Function for profiling.

p3.f

This is a revised version of the one in Chapter 8, “Debugging without the bug.

```
      real function determinant(m)
      real m(2,2)
      determinant = m(1,1) * m(2,2) - m(1,2) * m(2,1)
      return
      end
```

11.2 *The* `time` *Command*

The simplest way to gather data about the resources consumed by a program is to use the `time` (1) command, or, in `csh` to issue the “`set time`” command.

Example: Compile the sample program listed above (with or without `-g`), and run `time` on it. The output format may vary.

```
demo$ f77 -o silly -silent p1.f p2.f p3.f
Linking:
demo$ time silly
     1.00000
3.2u 0.3s 0:08 41% 0+104k 0+0io 0pf+0w
demo$ ■
```

Interpretation:

- 3.2 seconds on user code.
- 0.3 seconds executing system code on behalf of the user.

- 0 minutes and 8 seconds to complete.
- 41 percent (about) of machine's resources dedicated to this program.
- 0 kilobytes of program memory; 104 kilobytes of data memory (averages).
- 0 reads and 0 writes.
- 0 page faults.
- 0 swap-outs.

If there is some I/O, you get output similar to this.

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
```

Interpretation:

- 6 seconds on user code, approximately
- 17 seconds on system code on behalf of the user, approximately
- 1 minute 16 seconds to complete
- 31 per cent of the resources dedicated to this program
- 11 kilobytes of shared (program) memory
- 21 kilobytes of private (data) memory
- 354 reads
- 210 writes
- 135 page faults
- 0 swapouts

iMPact FORTRAN MP Notes

If iMPact FORTRAN MP is used, the number from `/bin/time` is interpreted in a different way. Since `/bin/time` accumulates the user time on different threads, the user number is no longer used, and only real time is used.

Since the user time displayed includes the time spent on all the processors, it can be quite large. So it is not a good measure of performance. The real time is the wall clock time and is a better measure of the performance.

Also since the real time is the wall clock time, if you run the parallel version of the benchmark, you might want to avoid running too many programs at the same time.

11

11.3 *The* `gprof` *Command*

The `gprof` (1) command provides a detailed procedure-by-procedure analysis of execution time, including how many times a procedure was called, who called it and who it called, and how much time was spent in the procedure and by the routines that it called.

Compile and Link

First, compile and link it with the `-pg` flag.

Example:

```
demo$ f77 -o silly -silent -pg p1.f p2.f p3.f
Linking:
demo$ ■
```

Execute

To get meaningful timing information, execution must complete normally.

```
demo$ silly
    1.00000
demo$ ■
```

After execution completes, a file named `gmon.out` gets written in the working directory. This file contains profiling data that can be interpreted with `gprof`.

Run `gprof`

Run the `gprof` program on the user program `silly`. The `gprof` program produces about 14 pages of report for this tiny program.

The report is mostly two profiles of how the total time is distributed across the program procedures:

- call graph
- flat profile.

They are preceded by an explanation of the column labels, and followed by an index.

In the following graph profile, the line that begins with "[4]" is called the *function line*, the lines above it the *parent lines*, and the lines below it the *descendant* lines.

Note – Only the first few lines of some sections are shown here.

Example: Selected `gprof` output:

```
demo$ gprof silly
     @(#)callg.blurb 1.5 88/02/08 SMI
 call graph profile:
    …
                                  called/total       parents
 index  %time    self descendents  called+self    name    index
                                  called/total       children

                                                     <spontaneous>
 [1]     99.5    0.00        3.82                 start [1]
                 0.00        3.82       1/1           _main [3]
                 0.00        0.00       1/1           _finitfp_ [303]
                 0.00        0.00       1/1           _on_exit [314]
 ---------------------------------------------
                 0.15        3.67       1/1           _main [3]
 [2]     99.5    0.15        3.67       1           _MAIN_ [2]
                 3.67        0.00  100000/100000      _mkidentity_ [4]
                 0.00        0.00       1/1           _s_wsle [317]
                 0.00        0.00       1/1           _determinant_ [296]
                 0.00        0.00       1/1           _do_l_out [297]
                 0.00        0.00       1/1           _e_wsle [299]
 ---------------------------------------------
                 0.00        3.82       1/1           start [1]
 [3]     99.5    0.00        3.82       1           _main [3]
                 0.15        3.67       1/1           _MAIN_ [2]
                 0.00        0.00      16/16          _signal [254]
                 0.00        0.00       1/1           _f_init [302]
              0.00       0.00      1/1          __enable_sigfpe_master [277]
               0.00       0.00       1/1           _ieee_retrospective_ [308]
                 0.00        0.00       1/1           _f_exit [301]
                 0.00        0.00       1/1           _exit [300]
 ---------------------------------------------
                 3.67        0.00  100000/100000      _MAIN_ [2]
 [4]     95.6    3.67        0.00  100000          _mkidentity_ [4]
 ---------------------------------------------
    …
demo$
```

The function line in the example above reveals that:

- `mkidentity` was called 100000 times.
- 3.67 seconds were spent in `mkidentity` itself,
- 0 seconds were spent in routines called by `mkidentity`.
- 95.6% of the execution time of `silly` is from `mkidentity`.

Parent line

The single parent line reveals that `MAIN` was the only procedure to call `mkidentity`, that is, all 100000 invocations of `mkidentity` came from `MAIN`. Thus, all of the 3.67 seconds spent in `mkidentity` were spent on behalf of `MAIN`. If `mkidentity` had also been called from another procedure, there would be two parent lines and the 3.67 seconds of *self* time would be divided between `MAIN` and the other caller. The descendant lines are interpreted similarly.

Overhead

When you enable profiling, the running time of a program may be significantly increased. The fact that `mcount`, the utility routine used to gather the raw profiling data, is usually at the top of the flat profile shows this. To eliminate this overhead in the completed version of the program, recompile all source files without the `-pg` flag. Ignore the overhead incurred by `mcount` when interpreting the flat profile. The graph profile attempts to automatically subtract time attributed to `mcount` when computing percentages of total runtime. This may not be accurate due to UNIX timekeeping conventions.

The FORTRAN library includes two routines that return the total time used by the calling process. See `dtime` (3F) and `etime` (3F).

11.4 The `tcov` *Command*

The `tcov` (1) command provides a detailed statement-by-statement profile of an actual test case of a program.

Compile and Link

First, compile and link it with the `-a` flag, as in this example. This example uses `-a` on all modules, but it is usually better to use the `-a` option on only those modules which profiling has shown to be most expensive.

```
demo$ f77 -silent -o silly -a p1.f p2.f p3.f
demo$ ■
```

Execute

To generate meaningful timing information, execution must complete normally, or the user code must call `exit(2)`.

```
demo$ silly
1.00000
demo$ ■
```

After execution completes, there is a new file named `p1.tcov` in the working directory. This file contains profiling data that can be interpreted with `tcov`.

Run `tcov`

Run the `tcov` program on the user source file `p1.f`.

```
demo$ tcov p1.f
demo$ ■
```

List `p1.tcov`

```
demo$ cat p1.tcov
                program silly
                parameter (n=2)
                real twobytwo(2,2) / 4 *-1 /
     1 ->     do i = 1, 100000
     100000 -> call mkidentity( twobytwo, n )
              end do
     1 ->       print *, determinant(twobytwo)
                end
           Top 10 Blocks

     Line        Count
     5           100000
     4                1
     7                1
     3           Basic blocks in this file
     3           Basic blocks executed
     100.00      Percent of the file executed
     100002      Total basic block executions
     33334.00    Average executions per basic block
demo$ ■
```

11.5 I/O Profiling

You can get a report about how much data was transferred by your program. For each FORTRAN unit, the report shows the file name, the number of I/O statements, the number of bytes, and some statistics on these items.

To get the I/O profiling report, do the following:

1. **Compile with the `-pg` option.**

```
demo$ f77 -pg src.f
```

2. **Insert the statement "`external start_iostats`" before the first executable statement, and insert a call to `start_iostats` before the first I/O statement that you want to measure.**

```
    external start_iostats
    …
    call start_iostats
```

   `stdin`, `stdout`, and `stderr`

   I/O statement includes `READ`, `WRITE`, `PRINT`, `OPEN`, `CLOSE`, `INQUIRE`, `BACKSPACE`, `ENDFILE`, and `REWIND`. The runtime system opens `stdin`, `stdout`, and `stderr` before the first executable statement of your program, so you must reopen these units after the call to `start_iostats` (without first closing them).

   Example: Profile `stdin`, `stdout`, and `stderr`.

```
    EXTERNAL start_iostats
    …
    CALL start_iostats
    OPEN(5)
    OPEN(6)
    OPEN(0)
```

   Call `end_iostats` to stop it, if you want to measure part of a program.

3. **Run your program.**

```
demo$ a.out
```

4. **Display the report file. If the executable file name is *name*, the report will be on the *name*.io_stats file.** Example:

```
demo$ cat a.out.io_stats
                Input Report
1. unit    2. file name              3. input data          4. map
                             cnt     total    avg   std dev    (cnt)
-------------------------------------------------------------------
      0          stderr      0         0    0.0      0.00    No
                             0         0    0.0      0.00
      5           stdin      0         0    0.0      0.00    No
                             0         0    0.0      0.00
      6          stdout      0         0    0.0      0.00    No
                             0         0    0.0      0.00
     10            temp      0         0    0.0      0.00    No
     …
               Output Report
1. unit            5. output data         6. blk size  7. fmt  8. direct
           cnt     total    avg   std dev                          (rec len)
-------------------------------------------------------------------------
      0      0         0    0.0      0.00           0     Yes     seq
             0         0    0.0      0.00
      5      0         0    0.0      0.00           0     Yes     seq
             0         0    0.0      0.00
      6      1         3    3.0      0.00           0     Yes     seq
             1         3    3.0      0.00
     10   2000      8000    4.0      0.00       16384     Yes     dir
     …
```

11.6 Missing Profile Libraries

If the profiling libraries are not installed, and if you try to use profiling, you may get a message like this:

```
demo$ f77 -p real.f
real.f:
 MAIN stuff:
ld: -lc_p: No such file or directory
demo$ ■
```

There is a system utility to extract files from the release CD. You can use it to get the debugging files after the system is installed. See add_services(8). You may want to get help from your system administrator.

Performance 12

This chapter is organized into the following sections.

Introduction	*page 225*
Why Tune Code?	*page 226*
Algorithm Choice	*page 226*
Tuning Methodology	*page 227*
Loop Jamming	*page 229*
Benchmark Case History	*page 230*
Optimizing	*page 234*

This chapter introduces performance and optimizing issues. Most references cited go into the subject far more deeply than this chapter.

12.1 Introduction

The following koans are provided to help establish the correct mind-set.

- There can be no cookbook for tuning.
- There is no substitute for experience and human cleverness. Many tactics can be (and must be) employed.
- The best I/O is no I/O.
- The best compute is no compute.

- Concentrate on the big picture. Solve the real problem.
- Don't forget details. A cycle here and a cycle there *in a key loop* adds up to many mips.
- Code tuning is not for the squeamish nor the faint of heart.
- It can be exciting—and frustrating.

12.2 Why Tune Code?

There are two situations where code tuning is vital.

- Benchmarking
- Application porting

12.3 Algorithm Choice

Algorithm choice is critical and is *always* made on the basis of machine architecture.

In olden times (1950-1970) all machines were scalar. Most were 32-60 bit, with extended precision accumulators. Memory was expensive. Therefore old algorithms were inner-product based (that is, dot-product based, like the Crout reduction, Cholesky Decomposition, and so forth). `sqrt` was expensive but improved numerical properties of the algorithms employed, so it allowed more problems to be run in single precision.

With the advent of the CRAY-1, vector algorithms became the rage. Dot products were replaced with SAXPY operations. New constraints on algorithms came about due to the difference between what computer scientists and mathematicians thought constituted a vector operation. In general, a vector algorithm does more work than a similar scalar algorithm.

Actually SAXPY became popular somewhat before the advent of vector machines. Dot product formulations tend to march through memory in the natural way (through the rows of each column) for one matrix, and the other way (through the columns of each row) for the other. On some high-performance scalar machines of that era, this resulted in suboptimal performance due to cache affects. SAXPY allows each matrix to be addressed in the natural fashion, at the cost of doing somewhat more memory accesses (and

losing some accuracy, due to the failure to accumulate in extended-precision registers). On vector machines, SAXPY is always the preferred technique because vector performance is really devastated by dot product formulations.

Example: Best incore sort: 300 < n < 700:
scalar ⟶ quicksort
vector ⟶ Pangali's bubble sort

The Pangali bubble sort does oodles more work, but executes 15 to 20 times faster on many vector machines.

Since most of us do not have vector machines, why worry about vector algorithms? One reason is that the user code might include attempts at complex vectorized algorithms. If you can replace complex vectorized code with simple scalar code, you can do less work and run faster. It can be much easier to concentrate on the underlying science and worry less about the programming.

12.4 Tuning Methodology

Get the program to run and to generate correct answers. Do not apply any tricks until you have correct results.

If the run is long (say > 20 minutes) and it is obvious how to reduce the problem size, do it. Rerun and save the output to be used as correct. If the code runs for a very long time (many hours), you must do this. If it runs for 22 minutes and changing the problem is not easy, skip to the next step.

Examine the profile. Recompile the top routines (say 80% of total time) with the `-a` switch to enable `tcov` analysis. Also toss in the `-pg` switch to count the number of calls, and time spent in the routine. Don't throw away the optimized (good) versions. You may have uses for them.

Rerun. It will take a little longer. Do not bother to obtain a *dry* machine, as this will not matter.

Run `gprof` and `tcov` (check the `man` pages, or the chapter on profiling).

Compare the `gprof` with the regular run — have the routines maintained their relative order? If so, continue without reservation. If not, work on routines in the order of their original import.

Consider subroutine COSTSaLOT.

```
      subroutine COSTSaLOT(randvec,n)
      real randvec(n)
      do i = 1, n
           randvec(i) = random() ! user random no. generator
      end do
      return
      end
```

tcov shows us the following:

```
           subroutine COSTSaLOT(randvec,n)
           real randvec(n)

       1 ->  do i = 1, n
 1000000 ->      randvec(i) = random()
             end do
       1 ->  return
             end

           real function random()

 1000000 -> random = rand(0)
           return
           end
```

The trick here is to *inline* the random number generator; that is, rewrite the program as the following.

```
      subroutine COSTSaLOT(randvec,n)
      real randvec(n)
      do i = 1, n
           randvec(i) = rand(0)
      end do
      return
      end
```

On a vector machine it is generally better to inline the code of rand itself, and then this is close to optimal. On SPARCsystems it may be better *not* to compute a whole vector at a time. Since there is have limited cache, it may be better to remove COSTSaLOT entirely, and simply call rand(0) from the calling program.

Lessons:

- `tcov` is handy for pinpointing exactly where to work.
- Try to inline small subroutines to reduce call overhead.
- Thinking that the best solution is to precompute a lot of things, does not make it so.

12.5 Loop Jamming

Start with a double loop like the following.

```
do i = 1, n
     stuff
end do
do i = 1, n
     more stuff
end do
```

You can rewrite it like this.

```
do i = 1, n
     stuff
     more stuff
end do
```

This can be a fair win on SPARCsystems. *It can also be a loss.* It will depend on cache sizes, SCRAM[1], and other considerations.

Lesson: Time it. Try it. If it wins, good. Lesson: Always take the loop in question and run it in isolation. Experimentation works. But *concentrate* on the loops that consume the most time. It will often be necessary to run some profiler program on the code.

This list only scratches the surface. Once you have narrowed down the expensive sections, it is easy to ask for assistance.

Do not forget to think about the algorithm — are you computing the best way?

1. SCRAM is on only the Sun-4/110, and stands for Static Column Random Access Memory.

12.6 Benchmark Case History

Consider the following trigonometric function benchmark.

test.f

```
      program test
      integer*4 limit, i, n
      parameter (limit=100000)
      double precision hold(3,limit), x1, x2, x3
      do 10 i = 1, limit
           do 5 n = 1, 3
                hold(n,i) = 0.0
5          continue
10    continue
      x1 = 0.0
      x2 = 0.0
      x3 = 0.0
      open( 3, file='test.tmp', form='FORMATTED' )
      do 20 i = 1, limit
           x1 = x1 + 1
           hold(1,i) = x1
           x2 = sin(hold(1,i)) - cos(hold(1,i))
           hold(2,i) = x2
           x3 = sqrt(hold(1,i)**2 + hold(2,i)**2)
           hold(3,i) = x3
           write(3,*) (hold(n,i),n = 1, 3)
           if (  x3 .le. 0.00001d0 .and.
                x3 .ge. -0.00001d0 ) then
              write(3,*) 'x3 = 0.0'
           elseif ( x2 .le. 0.00001d0 .and.
                   x2 .ge. -0.00001d0 ) then
              write(2,*) 'x2 = 0.0'
           else
                x2 = atan(x2/x3)
                x1 = hold(2,i) * hold(3,i)
           endif
20    continue
      close(3)
      end
```

This is one minute too slow, compared to some particular other computer; so recompile with the `-p` profiling option, and then profile the code with the `prof` utility. You can also use `-pg` and `gprof`. Examples are shown in the profiling chapter.

```
demo$ f77 -p -O3 test.f
test.f:
 AIN test:
demo$ prof a.out
```

Output from `prof`.

```
time a.out
real    4m19.36s
user    4m1.00s
sys     0m4.05s
prof
 %time  cumsecs  #call  ms/call  name
  24.0    58.32                   mcount
  10.2    83.10 499995     0.05  __fp_rightshift
   7.8   102.15 100000     0.19  _s_wsle
   7.3   119.96                .div
   7.2   137.42     64   272.81.rem
   4.7   148.874600144     0.00.umul
   4.0   158.65 300000     0.03  _unpacked_to_decimal
   3.9   168.09 300000     0.03  _wrt_F
   3.8   177.24 300000     0.03  __fp_leftshift
   3.0   184.44 300000     0.02  _fconvert
   2.6   190.881499992     0.00  __fourdigits
   2.6   197.28 300000     0.02  _binary_to_decimal_fraction
   2.4   203.081199996     0.00  __mul_10000
   1.6   207.01 299996     0.01  _binary_to_decimal_integer
   1.6   210.886299964     0.00.urem
   1.3   213.95                   _sincos
   1.1   216.57                   _MAIN_
   1.1   219.18 299996     0.01  __fp_normalize
   1.1   221.77                   _nwrt_A
      ... many more lines ...
   0.0   243.24      1     0.00  _strcpy
   0.0   243.24      1     0.00  _strlen
   0.0   243.24      3     0.00  _t_runc
$
```

What can you tell from this profile?

> `mcount` is sucking up much of the CPU. Therefore the program spends more time jumping between modules than computing. But the user code is very simple. Double checked that optimization was high (O3/4) and that in-lining was turned on (it was). So where is the time going?
>
> Note that the top routines are `.mul`, `.div`, `.rightshift`, and the user code isn't doing that. Furthermore, `sincos`, which is usually one of the most used routines, accounts for only 2% of the runtime.
>
> From this you can infer that something else (aside from trigonometric calculations) is going on. The output file shows this.

```
-rw-r--r-- 1 khb 5800000 Jan 25 13:02 test.tmp
```

> Note that it is quite large.
>
> Now examine the code, and note that *every* time through the loop it writes to the output file. If the program were large, you might have to recompile with `-a`, and run `tcov` to catch this.
>
> Modify the code. Eliminate not only the write, but the bogus `if` tests (comment them out in an obvious way).

```
cccccccc write(3,*) (hold(n,i),n = 1, 3)
cccccccc if (   x3 .le. 0.00001d0 .and.
cccccccc        x3 .ge. -0.00001d0 ) then
cccccccc    write(3,*) 'x3 = 0.0'
cccccccc elseif ( x2 .le. 0.00001d0 .and.
cccccccc          x2 .ge. -0.00001d0 ) then
cccccccc    write(2,*) 'x2 = 0.0'
           x2 = atan(x2/x3)
           x1 = jold(2,i) * hold(3,i)
cccccccc endif
```

Why *bogus*? Because on any IEEE machine, such as a SPARCsystem, there is no difficulty in computing x/0.0. It is ±Inf or 0.0/0.0 (NaN), or a bad `atan`. In real applications, you can do a large chain of operations and only need to check the final result (using `libm_single` and `libm_double` routines for IEEE handling, and perhaps `ieee_flags` to condition the exception flags). This can remove *millions* of `if` tests; that is, one `if` test in a key loop gets executed *millions* of times.

When you do this you get this output from `prof`.

```
real    0m9.65s           profiled times
user    0m4.35s
sys     0m0.60s

 %time  cumsecs  #call  ms/call  name
  62.3     2.71                  _sincos {much more sensible!}
  36.6     4.30                  _MAIN_
   0.5     4.32                  _cos
   0.5     4.34                  _sin
   0.2     4.35      5     2.00  _ioctl
   0.0     4.35     64     0.00 .rem
   0.0     4.35      1     0.00 .udiv
   0.0     4.35      3     0.00 .umul
   0.0     4.35      1     0.00  __enable_sigfpe_master
   0.0     4.35      1     0.00  __findiop
   0.0     4.35      1     0.00  _access
   0.0     4.35      2     0.00  _bzero
   0.0     4.35      2     0.00  _calloc
   0.0     4.35      4     0.00  _canseek
   0.0     4.35      4     0.00  _close
   0.0     4.35      1     0.00  _exit

      ... many more lines ...

   0.0     4.35      1     0.00  _strcpy
   0.0     4.35      1     0.00  _strlen
   0.0     4.35      2     0.00  _t_runc
$
```

Run it again without profiling, for optimal reporting time.

```
real 0m5.20s
user 0m4.28s
sys  0m0.61s
```

12

Lessons

- Performance analysis and tuning are iterative processes. Think about what you can do differently:
  a. Compile options
  b. Profile, but profile carefully; don't jump to conclusions.
  c. If necessary use `prof` and `tcov` to tell what is really happening.
- If you can get a 20x speedup by changing the code, *do it*. The IEEE arithmetic is new enough that not everyone knows how to use it to good advantage. Knowing why it is good and what it is good for can really help. It can be the start of a commitment to state-of-the-art standards.

12.7 Optimizing

At optimization level `-O4`, the compiler inlines calls to functions and subroutines which are defined in the same file as the caller. Thus, the usual UNIX advice of splitting each function and subroutine into a separate file may adversely impact performance. It may require experimentation with collecting different modules in different files to achieve maximum performance.

Further Reading

Most references cited go into the subject far more deeply than this chapter.

FORTRAN Reference Manual, Sun Microsystems

Numerical Computation Guide, Sun Microsystems

Performance Tuning an Application, Sun Microsystems

Programming Pearls, by Jon Louis Bentley, Addison Wesley

More Programming Pearls, by Jon Louis Bentley, Addison Wesley

Writing Efficient Programs, by Jon Louis Bentley, Prentice Hall.

FORTRAN Optimization, by Michael Metcalf, Academic Press 1982.

Optimizing FORTRAN Programs, by C. F. Schofield Ellis Horwood Ltd., 1989.

A Guidebook to Fortran on Supercomputers, Levesque, Williamson, Academic Press, 1989.

C–FORTRAN Interface 13

This chapter is organized into the following sections.

Sample Interface	*page 235*
How to Use this Chapter	*page 236*
Getting It Right	*page 237*
FORTRAN Calls C	*page 245*
C Calls FORTRAN	*page 268*

Glendower: I can call spirits from the vasty deep.

Hotspur: Why, so can I, or so can any man;
But will they come when you do call for them?
Henry IV, Part I

13.1 Sample Interface

As an introductory example, a FORTRAN main calls a C function.

Samp.c

```
samp ( i, f )
     int *i;
     float *f;
{
     *i = 9;
     *f = 9.9;
}
```

Above, both `i` and `f` are pointers.

Sampmain.f

```
      integer i
      real r
      external Samp !$pragma C ( Samp )
      call Samp ( i, r )
      write( *, "(I2, F4.1)") i, r
      end
```

Above, both `i` and `f` are passed by reference (the default).

Compile and execute, with output.

```
demo$ cc -c Samp.c
demo$ f77 -silent Samp.o Sampmain.f
demo$ a.out
 9 9.9
demo$ ■
```

13.2 How to Use this Chapter

1. **Examine the above sample and the section "Getting It Right"**
2. **See the section "FORTRAN Calls C" or "C Calls FORTRAN"**
3. **Within that section, choose one of these subsections:**

- Arguments passed by reference
- Arguments passed by value
- Function return values
- Labeled common
- Sharing I/O
- Alternate returns

4. **Within that subsection, choose one of these examples:**

   For the arguments, there is an example for each of these, or a note that it cannot be done.

- Simple types (character*1, logical, integer, real, double precision, quad)
- Complex types (complex, double complex)
- Character strings (character*n)
- One-dimensional arrays (integer a(9))
- Two-dimensional arrays (integer a(4,4))

- Structured records (structure and record)
- Pointers

  For *function return values*, there is an example for each of these:

  - Integer (int)
  - Real (float)
  - Pointer to real (pointer to float)
  - Double precision (double)
  - Quadruple precision (long double)
  - Complex
  - Character string

  For each of *labeled common*, *sharing I/O*, and *alternate returns*, there is one set of examples. These are the same for "FORTRAN calls C" or" C calls FORTRAN."

13.3 Getting It Right

Most C/FORTRAN interfaces must get all of these aspects right:

- Function/subroutine: Definition and call
- Data types: Compatibility of types
- Arguments: Pass by reference or value
- Arguments: Order
- Procedure name: Upper and lowercase and trailing underscore (_)
- Libraries: Telling the linker to use FORTRAN libraries

Some C/FORTRAN interfaces must also get these right:

- Arrays: Indexing and order
- File descriptors and `stdio`
- File permissions

Function or Subroutine

The word *function* means different things in C and FORTRAN.

- As far as C is concerned, all subprograms are functions, it is just that some of them return a null value.
- As far as FORTRAN is concerned, a function passes a return value and a subroutine does not.

FORTRAN Calls a C Function

- If the called C function returns a value, call it from FORTRAN as a function.
- If the called C function does not return a value, call it as a subroutine.

C Calls a FORTRAN Subprogram

- If the called FORTRAN subprogram is a *function*, call it from C as a function that returns a comparable data type.
- If the called FORTRAN subprogram is a *subroutine*, call it from C as a function that returns a value of `int` (comparable to FORTRAN `INTEGER*4`) or `void`. This return value is useful if the FORTRAN routine does a nonstandard return.

Data Type Compatibility

Data types have the following sizes and alignments without `-f`, `-i2`, `-misalign`, `-r4`, or `-r8`.

Table 13-1 Argument Sizes and Alignments—Pass by Reference

FORTRAN Type	*C Type*	*Size (bytes)*	*Alignment (bytes)*
byte x	char x	1	1
character x	char x	1	1
character*n x	char x[n]	*n*	1
complex x	struct {float r,i;} x;	8	4
complex*8 x	struct {float r,i;} x;	8	4
double complex x	struct {double dr,di;}x;	16	4
complex*16 x	struct {double dr,di;}x;	16	4
complex*32 x	struct {long double dr,di;} x;	32	4
double precision x	double x	8	4
real x	float x	4	4
real*4 x	float x	4	4
real*8 x	double x	8	4
real*16 x	long double x	16	4
integer x	int x	4	4
integer*2 x	short x	2	2
integer*4 x	int x	4	4
logical x	int x	4	4
logical*4 x	int x	4	4
logical*2 x	short x	2	2
logical*1 x	char x	1	1

The `REAL*16` and the `COMPLEX*32` can be passed between FORTRAN and ANSI C, but not between FORTRAN and some previous versions of C.

`complex*32`, SPARC only

`real*16`, SPARC only

Remarks

- Alignments are for FORTRAN types.
- Arrays pass by reference, if the elements are compatible.
- Structures pass by reference, if the fields are compatible.
- Passing arguments by value:
  - You cannot pass arrays, character strings, or structures by value.
  - You can pass arguments by value from FORTRAN to C, but not from C to FORTRAN, since the `%VAL()` does not work in a `SUBROUTINE` statement.

Case Sensitivity

C and FORTRAN take opposite perspectives on case sensitivity.

- C is case sensitive—uppercase or lowercase matters.
- FORTRAN ignores case.

The FORTRAN default is to ignore case by converting subprogram names to lowercase. It converts all uppercase letters to lowercase letters, except within character-string constants.

There are two common solutions to the uppercase/lowercase problem.

- In the C subprogram, make the name of the C function be all lowercase.
- Compile the FORTRAN program with the `-U` option, which tells FORTRAN to preserve existing uppercase/lowercase distinctions (do not convert to all lowercase letters).

Use one or the other, but not both.

Most examples in this chapter use all lowercase letters for the name in the C function, and do *not* use the FORTRAN `-U` compiler option.

Underscore in Names of Routines

The FORTRAN compiler normally appends an underscore (_) to the names of subprograms, for both a subprogram and a call to a subprogram. This distinguishes it from C procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All FORTRAN library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are two common solutions to the underscore problem.

- In the C function, change the name of the function by appending an underscore to that name.
- Use the `C()` pragma to tell the FORTRAN compiler to omit those trailing underscores.

Use one or the other, but not both.

Most of the examples in this chapter use the FORTRAN `C()` compiler pragma and do *not* use the underscores. The `C()` pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C language, so the FORTRAN compiler does not append an underscore to such names, as it ordinarily does with external names. The `C()` directive for a particular function must appear before the first reference to that function. It must appear in each subprogram that contains such a reference. The conventional usage is this.

```
      EXTERNAL ABC, XYZ!$PRAGMA C( ABC, XYZ )
```

If you use this pragma, then in the C function you must *not* append an underscore to those names.

Passing Arguments by Reference or Value

In general, FORTRAN passes arguments by reference. In a call, if you enclose an argument with the nonstandard function `%VAL()`, FORTRAN passes it by value.

In general, C passes arguments by value. If you precede an argument by an ampersand (&), C passes it by reference. C always passes arrays and character strings by reference.

Arguments and Order

For every argument of character type, an argument is passed giving the length of the value. The string lengths are equivalent to C `long int` quantities passed by value.

The order of arguments is:

- Address for each argument (datum or function).
- A `long int` for each character argument (The whole list of string lengths comes after the whole list of other arguments.)

Example: A FORTRAN call in a code fragment.

```
      CHARACTER*7 S
      INTEGER B(3)
       …
      CALL SAM( B(2), S )
```

The above call is equivalent to the C call in this code fragment.

```
char s[7];
long b[3];
  …
sam_( &b[1], s, 7L ) ;
```

Array Indexing and Order

Array Indexing

C arrays always start at zero, but by default, FORTRAN arrays start at 1. There are two common ways of approaching this.

- You can use the FORTRAN default, as in the above example. Then the FORTRAN element `B(2)` is equivalent to the C element `b[1]`.
- You can specify that the FORTRAN array `B` starts at 0. as follows.

```
INTEGER B(0:2)
```

This way the FORTRAN element `B(1)` is equivalent to the C element `b[1]`.

Array Order

FORTRAN arrays are stored in column-major order, C arrays in row-major order. For one-dimensional arrays, this is no problem. For two-dimensional arrays, this is only a minor problem, as long as the array is square. Sometimes it is enough to just switch subscripts.

For two-dimensional arrays that are not square, it is not enough to just switch subscripts. For arrays of more than two dimensions, this is usually considered too much of a problem.

Libraries and Linking with the `f77` Command

To get the proper FORTRAN libraries linked, use the `f77` command to pass the `.o` files on to the linker. This usually shows up as a problem only if a C main calls FORTRAN. *Dynamic* linking is encouraged and made easy.

Example 1: Use f77 to link.

```
demo$ f77 -c -silent RetCmplx.f
demo$ cc -c RetCmplxmain.c
demo$ f77 RetCmplx.o RetCmplxmain.o ← This does the linking.
demo$ a.out
 4.0 4.5
 8.0 9.0
demo$ ■
```

Example 2: Use `cc` to link. This fails. The libraries are not linked.

```
demo$ f77 -silent -c RetCmplx.f
demo$ cc RetCmplx.o RetCmplxmain.c  ←  wrong link command
ld: Undefined symbol    ←  missing routine
 __Fc_mult
demo$ ■
```

File Descriptors and `stdio`

FORTRAN I/O channels are in terms of unit numbers. The I/O system does not deal with unit numbers, but with *file descriptors*. The FORTRAN runtime system translates from one to the other, so most FORTRAN programs don't have to know about file descriptors.

Many C programs use a set of subroutines called *standard I/O* (or `stdio`). Many functions of FORTRAN I/O use standard I/O, which in turn uses operating system I/O calls. Some of the characteristics of these I/O systems are listed below.

Table 13-2 Characteristics of Three I/O Systems

	FORTRAN Units	***Standard I/O File Pointers***	***File Descriptors***
Files Open	Opened for reading and writing	Opened for reading; or Opened for writing; or Opened for both; or Opened for appending See `OPEN`(3S).	Opened for reading; or Opened for writing; or Opened for both
Attributes	Formatted or unformatted	Always unformatted, but can be read or written with format-interpreting routines	Always unformatted
Access	Direct or sequential	Direct access if the physical file representation is direct access, but can always be read sequentially	Direct access if the physical file representation is direct access, but can always be read sequentially
Structure	Record	Character stream	Character stream
Form	Arbitrary nonnegative integers	Pointers to structures in the user's address space	Integers from 0-63

File Permissions

C programmers traditionally open input files for reading and output files for writing, sometimes for both. In FORTRAN it's not possible for the system to foresee what use you will make of the file since there's no parameter to the `OPEN` statement that gives that information.

FORTRAN tries to open a file with the maximum permissions possible, first for both reading and writing then for each separately.

This occurs transparently and is of concern only if you try to perform a `READ`, `WRITE`, or `ENDFILE` but you don't have permission. Magnetic tape operations are an exception to this general freedom, since you can have write permissions on a file but not have a write ring on the tape.

13.4 FORTRAN Calls C

Arguments Passed by Reference (f77 Calls C)

Simple Types Passed by Reference (f77 Calls C)

For simple types, define each C argument as a pointer.

SimRef.c

```
simref ( t, f, c, i, r, d, q, si )
   char    * t, * f, * c ;
   int     * i ;
   float   * r ;
   double  * d ;
   long double  * q ;
   short   * si ;
{
        *t = 1 ; *f = 0 ;
        *c  = 'z' ;
        *i  = 9 ;
        *r  = 9.9 ;
        *d  = 9.9 ;
        *q  = 9.9 ;
        *si = 9 ;
}
```

Default: Pass each FORTRAN argument by reference.

SimRefmain.f

real*16, SPARC only

```
        logical*1  t, f
        character  c
        integer    i*4, si*2
        real       r*4, d*8, q*16
        external SimRef !$pragma C( SimRef )
        call SimRef ( t, f, c, i, r, d, q, si )
        write(*, "(L2,L2,A2,I2,F4.1,F4.1,F4.1,I2)")
&                  t, f, c, i, r,   d,   q,   si
        end
```

Compile and execute, with output.

```
demo$ cc -c SimRef.c
demo$ f77 -silent SimRef.o SimRefmain.f
demo$ a.out
 T F z 9 9.9 9.9 9.9 9
demo$ ■
```

Complex Types Passed by Reference (`f77` Calls C)

Here the C argument is a pointer to a structure.

CmplxRef.c

```
cmplxref ( w, z )
     struct complex { float r, i; } *w;
     struct dcomplex { double r, i; } *z;
{
     w -> r = 6;
     w -> i = 7;
     z -> r = 8;
     z -> i = 9;
}
```

CmplxRefmain.f

```
      complex w
      double complex z
      external CmplxRef !$pragma C( CmplxRef )
      call CmplxRef ( w, z )
      write(*,*) w
      write(*,*) z
      end
```

Compile and execute, with output.

```
demo$ cc -c CmplxRef.c
demo$ f77 -silent CmplxRef.o CmplxRefmain.f
demo$ a.out
  ( 6.00000,   7.00000)
  ( 8.0000000000000,   9.0000000000000)
demo$ ■
```

Character Strings Passed by Reference (`f77` Calls C)

Passing strings between C and FORTRAN is not encouraged.

Rules for Passing Strings

- All C strings pass by reference.
- For each FORTRAN argument of character type, an *extra argument* is passed giving the length of the string. The extra argument is equivalent to a C `long int` passed by value. This is nonstandard.
- The order of arguments is as follows:
  - A list of the regular arguments
  - A list of lengths, one for each character argument, each as a `long int`
  - The list of extra arguments comes after the list of regular arguments.

Example: Character strings passed by reference. A FORTRAN call.

```
      CHARACTER*7 S
      INTEGER B(3)
…
      CALL SAM( B(2), S )
```

The above call is equivalent to the C call in the following.

```
      char s[7];
      long b[3];
…
      sam_( &b[1], s, 7L );
```

Ignoring the Extra Arguments of Passed Strings

You can *ignore* the extra arguments, since they are after the list of other arguments. The following C function *ignores* the extra arguments.

StrRef.c

```
strref ( s10, s80 )
      char *s10, *s80;
{
      static char ax[11] = "abcdefghij";
      static char sx[81] = "abcdefghijklmnopqrstuvwxyz";
      strncpy ( s10, ax, 11 );
      strncpy ( s80, sx, 26 );
}
```

The following FORTRAN call generates hidden extra arguments.

StrRefmain.f

```
      character s10*10, s80*80
      external StrRef !$pragma C( StrRef )
      call StrRef( s10, s80 )
      write (*, 1) s10, s80
1     format("s10='", A, "'", / "s80='", A, "'")
      end
```

Compile and execute, with output.

```
demo$ cc -c StrRef.c
demo$ f77 -silent StrRef.o StrRefmain.f
demo$ a.out
s10='abcdefghij'
s80='abcdefghijklmnopqrstuvwxyz'
demo$ ■
```

Using the Extra Arguments of Passed Strings

You can *use* the extra arguments.

The following C function *uses* the extra arguments. It prints the lengths; what you really do with them is up to you.

StrRef2.c

```
strref ( s10, s80, L10, L80 )
     char *s10, *s80 ;
     long L10, L80 ;
{
     static char ax[11] = "abcdefghij" ;
     static char sx[81] = "abcdefghijklmnopqrstuvwxyz" ;
     printf("%d %d \n", L10, L80 ) ;
     strncpy ( s10, ax, 11 ) ;
     strncpy ( s80, sx, 26 ) ;
}
```

If you compile `StrRef2.c` and `StrRefmain.f`, then you get this output.

```
10 80
s10='abcdefghij'
s80='abcdefghijklmnopqrstuvwxyz'
```

One-Dimensional Arrays Passed by Reference (f77 Calls C)

A C array, indexed from 0 to 8.

FixVec.c

```
fixvec ( V, Sum )
     int *Sum;
     int V[9];
{
     int i;
     *Sum = 0;
     for ( i = 0; i <= 8; i++ ) *Sum = *Sum + V[i];
}
```

A FORTRAN array, implicitly indexed from 1 to 9.

FixVecmain.f

```
     integer i, Sum
     integer a(9) / 1,2,3,4,5,6,7,8,9 /
     external FixVec !$pragma C( FixVec )
     call FixVec ( a, Sum )
     write(*, '(9I2, " ->" I3)') (a(i),i=1,9), Sum
     end
```

Compile and execute, with output.

```
demo$ cc -c FixVec.c
demo$ f77 -silent FixVec.o FixVecmain.f
demo$ a.out
 1 2 3 4 5 6 7 8 9 -> 45
demo$ ■
```

A FORTRAN array, explicitly indexed from 0 to 8.

FixVecmain2.f

```
     integer i, Sum
     integer a(0:8) / 1,2,3,4,5,6,7,8,9 /
     external FixVec !$pragma C( FixVec )
     call FixVec ( a, Sum )
     write(*, '(9I2, " ->" I3)') (a(i),i=0,8), Sum
     end
```

Compile and execute, with output.

```
demo$ cc -c FixVec.c
demo$ f77 -silent FixVec.o FixVecmain2.f
demo$ a.out
 1 2 3 4 5 6 7 8 9 -> 45
demo$ ■
```

Two-Dimensional Arrays Passed by Reference (f77 Calls C)

In a two-dimensional array, the rows and columns are switched. Such square arrays are either incompatible between C and FORTRAN, or awkward to keep straight, depending on your attitude or needs. Nonsquare arrays are worse.

Example: A 2 by 2 C array, indexed from 0 to 1, and 0 to 1.

FixMat.c

```
fixmat ( a )
     int a[2][2];
{
     a[0][1] = 99;
}
```

A 2 by 2 FORTRAN array, explicitly indexed from 0 to 1, and 0 to 1.

FixMatmain.f

```
      integer c, m(0:1,0:1) / 00, 10, 01, 11 /, r
      external FixMat !$pragma C ( FixMat )
      do r = 0, 1
           do c = 0, 1
              write(*,'("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
           end do
      end do
      call FixMat ( m )
      do r = 0, 1
           do c = 0, 1
              write(*,'("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
           end do
      end do
      end
```

Compile and execute. Show `m` before and after the C call.

Compare `a[0][1]` with `m(1,0)`: C changed `a[0][1]`, which is FORTRAN `m(1,0)`.

```
demo$ cc -c FixMat.c
demo$ f77 -silent FixMat.o FixMatmain.f
demo$ a.out
m(0,0) = 00
m(0,1) = 01
m(1,0) = 10
m(1,1) = 11
m(0,0) = 00
m(0,1) = 01
m(1,0) = 99
m(1,1) = 11
demo$ ■
```

Structured Records Passed by Reference (`f77` Calls C)

Example: A C structure of an integer and a character string.

`StruRef.c`

```
struct VarLenStr {
     int nbytes;
     char a[26];
};
void struchr ( v )
struct VarLenStr *v;
{
     bcopy( "oyvay", v->a, 5 );
     v->nbytes = 5;
}
```

A FORTRAN structured record of an integer and a character string.

StruRefmain.f

```
      structure /VarLenStr/
           integer nbytes
           character a*25
      end structure
      record /VarLenStr/ vls
      character s25*25
      external StruChr !$pragma C( StruChr )
      vls.nbytes = 0
      Call StruChr( vls )
      s25(1:5) = vls.a(1:vls.nbytes)
      write ( *, 1 ) vls.nbytes, s25
1     format("size =", I2, ", s25='", A, "'" )
      end
```

Compile and execute, with output.

```
demo$ cc -c StruRef.c
demo$ f77 -silent StruRef.o StruRefmain.f
demo$ a.out
size = 5, s25='oyvay'
demo$ ■
```

Pointers Passed by Reference (`f77` Calls C)

C gets it as a pointer to a pointer.

PassPtr.c

```
passptr ( i, d )
     int **i;
     double **d;
{
     **i = 9;
     **d = 9.9;
}
```

FORTRAN passes by reference, and it is passing a pointer.

PassPtrmain.f

```
      program PassPtrmain
      integer          i
      double precision d
      pointer ( iPtr, i ), ( dPtr, d )
      external PassPtr !$pragma C( PassPtr )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtr ( iPtr, dPtr )
      write( *, "(i2, f4.1)" )  i, d
      end
```

Compile and execute, with output.

```
demo$ cc -c PassPtr.c
demo$ f77 -silent PassPtr.o PassPtrmain.f
demo$ a.out
 9 9.9
demo$ ■
```

Arguments Passed by Value (f77 Calls C)

In the call, enclose an argument in the nonstandard function %VAL(). This works for all simple types and pointers.

Simple Types Passed by Value (f77 Calls C)

If you prototype the float parameter, C does not promote to double.

SimVal.c

```
simval ( char t, char c, int i, float r, double d,
     long double q, short s, int *reply )
{
        *reply = 0 ;
        /* If nth arg ok, set nth octal digit to one */
        if ( t         ) *reply = *reply + 1 ;
        if ( c == 'z'  ) *reply = *reply + 8 ;
        if ( i == 9    ) *reply = *reply + 64 ;
        if ( r == 9.9F ) *reply = *reply + 512 ;
        if ( d == 9.9  ) *reply = *reply + 4096 ;
        if ( q == 9.9L ) *reply = *reply + 32768 ;
        if ( s == 9    ) *reply = *reply + 262144 ;
}
```

Pass each FORTRAN argument by value, except for args.

SimValmain.f

real*16, SPARC only

```
        logical*1  t
        character  c
        integer    i*4, s*2, args*4
        real       r*4, d*8, q*16
        data t / .true. /,  c / 'z' /
&            i/ 9 /,  r/9.9/, d/ 9.9D0 /,  q/ 9.9Q0 /,  s/ 9 /
        external SimVal !$pragma C( SimVal )
        call SimVal( %VAL(t), %VAL(c), %VAL(i),
&                    %VAL(r), %VAL(d), %VAL(q), %VAL(s), args )
        write( *, 1 )  args
 1      format('args=', o7, '(If nth digit=1, arg n OK)')
        end
```

Compile and execute, with output.

```
demo$ cc -c SimVal.c
demo$ f77 -silent SimVal.o SimValmain.f
demo$ a.out
args=1111111(If nth digit=1, arg n OK)
demo$ ■
```

Complex Types Passed by Value (f77 Calls C)

You can pass the `complex` structure by value.

CmplxVal.c

```
cmplxval ( w, z )
     struct complex { float r, i; } w, *z;
{
     z->r = w.r * 2.0;
     z->i = w.i * 2.0;
     w.r = 0.0;
     w.i = 0.0;
}
```

CmplxValmain.f

```
     complex w / ( 4.0, 4.5 ) /
     complex z
     external CmplxVal !$pragma C( CmplxVal )
     call CmplxVal ( %VAL(w), z )
     write(*,*) w
     write(*,*) z
     end
```

Compile and execute, with output.

```
demo$ cc -c CmplxVal.c
demo$ f77 -silent CmplxVal.o CmplxValmain.f
demo$ a.out
  ( 4.00000, 4.50000)
  ( 8.00000, 9.00000)
demo$ ■
```

Arrays, Strings, Structures Passed by Value (f77 Calls C) - N/A

There is no way at all to pass arrays, character strings, or structures by value—at least not a reliable way that works on all architectures. The work-around for these is to pass them by reference.

Pointers Passed by Value (f77 Calls C)

C receives the argument as a pointer.

PassPtrVal.c

```
passptrval ( i, d  )
     int     *i ;
     double   *d ;
{
     *i  = 9 ;
     *d = 9.9 ;
}
```

FORTRAN passes a pointer by value.

PassPtrValmain.f

```
      program PassPtrValmain
      integer          i
      double precision d
      pointer ( iPtr, i ), ( dPtr, d )
      external PassPtrVal !$pragma C( PassPtrVal )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtrVal ( %VAL(iPtr), %VAL(dPtr) ) ! Nonstandard
      write( *, "(i2, f4.1)" )  i, d
      end
```

Compile and execute, with output.

```
demo$ cc -c PassPtrVal.c
demo$ f77 -silent PassPtrVal.o PassPtrValmain.f
demo$ a.out
 9 9.9
demo$ ■
```

Function Return Values (f77 Calls C)

For function return values, a FORTRAN function of type BYTE, INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or REAL*16 (quadruple precision) is equivalent to a C function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

Return an int (f77 Calls C)

RetInt.c

```
int retint ( r )
int *r;
{
     int s;
     s = *r;
     s++;
     return ( s );
}
```

RetIntmain.f

```
      integer r, s, RetInt
      external RetInt !$pragma C( RetInt )
      r = 8
      s = RetInt ( r )
      write( *, "(2I4)") r, s
      end
```

Compile, link, and execute, with output.

```
demo$ cc -c RetInt.c
demo$ f77 -silent RetInt.o RetIntmain.f
demo$ a.out
 8 9
demo$
```

In the same way, do a function of type BYTE, LOGICAL, REAL, or DOUBLE PRECISION. Use matching types according to Table 13-1.

Return a `float` *(*`f77` *Calls C)*

RetFloat.c

```
float  retfloat ( pf )
float *pf ;
{
     float  f ;
     f = *pf ;
     f++ ;
     return ( f ) ;
}
```

RetFloatmain.f

```
     real  RetFloat, R, S
     external RetFloat !$pragma C( RetFloat )
     R = 8.0
     S = RetFloat ( R )
     print *, R, S
     end
```

```
demo$ cc -c RetFloat.c
demo$ f77 -silent RetFloat.o RetFloatmain.f
demo$ a.out
    8.00000 9.00000
demo$ ■
```

In earlier versions of C, if C returned a function value that was a `float`, C promoted it to a `double`, and various tricks were needed to get around that.

Return a Pointer to a `float` *(*`f77` *Calls C)*

This example shows how to return a function value that is a pointer to a `float`. Compare with previous example.

RetPtrF.c

```
static float f;
float *retptrf ( a )
float *a;
{
     f = *a;
     f++;
     return &f;
}
```

RetPtrFmain.f

```
      integer RetPtrF
      external RetPtrF !$pragma C( RetPtrF )
      pointer ( P, S )
      real R, S
      R = 8.0
      P = RetPtrF ( R )
      print *, S
      end
```

Compile and execute, with output.

```
demo$ cc -c RetPtrF.c
demo$ f77 -silent RetPtrF.o RetPtrFmain.f
demo$ a.out
9.00000
demo$ ■
```

Since the function return value is an address, you can assign it to the pointer value, or possibly do some pointer arithmetic. You *cannot* use it in an expression with, say, reals, such as `RetPtrF(R)+100.0`.

Return a `DOUBLE PRECISION` *(*`f77` *Calls C)*

Here is an example of C returning a type `double` function value to a FORTRAN `DOUBLE PRECISION` variable.

RetDbl.c

```
double retdbl ( r )
double *r;
{
      double s;
      s = *r;
      s++;
      return ( s );
}
```

RetDblmain.f

```
      double precision r, s, RetDbl
      external RetDbl !$pragma C( RetDbl )
      r = 8.0
      s = RetDbl ( r )
      write( *, "(2F6.1)") r, s
      end
```

Compile and execute, with output.

```
demo$ cc -c RetDbl.c
demo$ f77 -silent RetDbl.o RetDblmain.f
demo$ a.out
   8.0 9.0
demo$ ■
```

Return a Quadruple Precision (`f77` Calls C)

Example: C returns a `long double` to a FORTRAN `REAL*16`.

RetQuad.c

SPARC only

```
long double  retquad ( pq )
long double *pq ;
{
     long double  q ;
     q = *pq ;
     q++ ;
     return ( q ) ;
}
```

RetQuadmain.f

SPARC only

```
      real*16  RetQuad, R, S
      external RetQuad !$pragma C( RetQuad )
      R = 8.0
      S = RetQuad ( R )
      write(*,'(2F6.1)') R, S
      end
```

Compile and execute, with output.

```
demo$ cc -c RetQuad.c
demo$ f77 -silent RetQuad.o RetQuadmain.f
demo$ a.out
   8.0   9.0
demo$ ■
```

Return a COMPLEX *(*f77 *Calls C)*

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is the following:

```
      COMPLEX FUNCTION F (…)
```

The pattern for a corresponding C function is

```
f_ (temp, … )
struct { float r, i; } *temp;
```

Example: C returns a type COMPLEX function value to FORTRAN.

RetCmplx.c

```
struct complex { float r, i; };
void retcmplx ( temp, w )
struct complex *temp;
struct complex *w;
{
     temp->r = w->r + 1.0;
     temp->i = w->i + 1.0;
     return;
}
```

RetCmplxmain.f

```
      complex u, v, RetCmplx
      external RetCmplx !$pragma C( RetCmplx )
      u = ( 7.0, 8.0 )
      v = RetCmplx ( u )
      write( *, * ) u
      write( *, * ) v
      end
```

Compile and execute, with output.

```
demo$ cc -c -silent RetCmplx.c
demo$ f77 -silent RetCmplx.o RetCmplxmain.f
demo$ a.out
  ( 7.00000, 8.00000)
  ( 8.00000, 9.00000)
demo$ ■
```

Return a Character String (`f77` Calls C)

Passing strings between C and FORTRAN is not encouraged. A character-string-valued FORTRAN function is equivalent to a C function with the two extra initial arguments — data address and length.

A FORTRAN function of this form, with no `C()` pragma is the following:

```
      CHARACTER*15 FUNCTION G ( … )
```

The above FORTRAN function is equivalent to a C function of this form.

```
g_ ( result, length, … )
char result[ ];
long length;
```

In either form, the function can be invoked in C with this call.

```
char chars[15];
…
      g_ ( chars, 15L, … );
```

Example. No pragma.

RetStr.c

```
retstr_ ( retval_ptr, retval_len, ch_ptr, n_ptr, ch_len )
char *retval_ptr, *ch_ptr;
int retval_len, *n_ptr, ch_len;
{
     int count, i;
     char *cp;
     count = *n_ptr;
     cp = retval_ptr;
     for (i=0; i<count; i++) {
          *cp++ = *ch_ptr;
     }
}
```

In the above example, note the following:

- The returned string is passed by the extra arguments `retval_ptr` and `retval_len`, a pointer to the start of the string and the string's length.
- The character-string argument is passed with `ch_ptr` and `ch_len`.
- The `ch_len` is at the end of the argument list.
- The repeat factor is passed as `n_ptr`.

In FORTRAN, use the above C function from `RetStr.c` as follows.

RetStrmain.f

```
     CHARACTER String*100, RetStr*50
     String = RetStr ( '*', 10 )
     PRINT *, "'", String(1:10), "'"
     END
```

The output from `RetStrmain.f` is as follows.

```
demo$ cc -c RetStr.c
demo$ f77 -silent RetStr.o RetStrmain.f
demo$ a.out
'**********'
demo$ ■
```

13

Labeled Common (f77 Calls C)

C and FORTRAN can share values in labeled common. The method is the same no matter which language calls which.

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common / ilk / u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

UseCommain.c

```
#include <stdio.h>
extern struct comtype {
     float p;
     float q;
     float r;
};
extern struct comtype ilk_;
main()
{
     char *string = "abc0";
     int count = 3;
     extern void usecom_ ( );
     ilk_.p = 1.0;
     ilk_.q = 2.0;
     ilk_.r = 3.0;
     usecom_ ( string, count );
     printf(" ilk_.p=%4.1f, ilk_.q=%4.1f, ilk_.r=%4.1f\n",
     ilk_.p, ilk_.q, ilk_.r );
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent UseCom.f
demo$ cc -c UseCommain.c
demo$ f77 UseCom.o UseCommain.o
demo$ a.out
 ilk_.p = 7.0, ilk_.q = 8.0, ilk_.r = 9.0
demo$ ■
```

Any of the options that change size or alignment (or any equivalences that change alignment) might invalidate such sharing.

Sharing I/O (`f77` Calls C)

Mixing FORTRAN I/O with C I/O is not recommended. If you must mix them, it is usually safer to pick one and stick with it, rather than alternating.

The FORTRAN I/O library is implemented largely on top of the C standard I/O library. Every open unit in a FORTRAN program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

If a FORTRAN main program calls C, then before the FORTRAN program starts, the FORTRAN I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C function must take the FORTRAN I/O environment into consideration to perform I/O on open file descriptors.

Mixing with `stdout` (`f77` Calls C)

A C function that writes to `stderr` and to `stdout` is shown below.

MixIO.c

```
#include <stdio.h>
mixio ( n )
int *n;
{
     if ( *n <= 0 ) {
          fprintf ( stderr, "error: negative line #\n" );
          *n = 1;
     }
     printf ( "In C: line # = %d \n", *n );
}
```

In FORTRAN, use the above C function as follows.

MixIOmain.f

```
      integer n / -9 /
      external MixIO !$pragma C( MixIO )
      do i = 1, 6
           n = n + 1
           if ( abs(mod(n,2)) .eq. 1 ) then
                call MixIO ( n )
           else
                write(*,'("In FORTRAN: line # =",i2)') n
           end if
      end do
      end
```

Compile and execute, with output.

```
demo$ cc -c MixIO.c
demo$ f77 -silent MixIO.o MixIOmain.f
demo$ a.out
In FORTRAN: line # =-8
error: negative line #
In C: line # = 1
In FORTRAN: line # = 2
In C: line # = 3
In FORTRAN: line # = 4
In C: line # = 5
demo$ ■
```

Mixing with `stdin` *(*`f77` *Calls C)*

A C function that reads from `stdin` is shown below.

MixStdin.c

```
#include <stdio.h>
int c_read_ ( fd, buf, nbytes, buf_len )
FILE **fd;
char *buf;
int *nbytes, buf_len;
{
     return fread ( buf, 1, *nbytes, *fd );
}
```

In FORTRAN, use the above C function as follows.

MixStdinmain.f

```
      character*1 inbyte
      integer*4 c_read, getfilep
      external getfilep
      write(*,'(a,$)') 'What is the digit? '
      irtn = c_read ( getfilep(5), inbyte, 1 )
      write(*,9) inbyte
9     format('The digit read by C is ', a )
      end
```

FORTRAN does the prompt. C does the read.

```
demo$ cc -c MixStdin.c
demo$ f77 -silent MixStdin.o MixStdinmain.f
demo$ a.out
What is the digit? 3
The digit read by C is 3
demo$
```

Alternate Returns (f77 Calls C) - N/A

C does not have an alternate return. The work-around is to pass an argument and branch on that.

13.5 C Calls FORTRAN

Arguments Passed by Reference (C Calls f77)

Simple Types Passed by Reference (C Calls f77)

FORTRAN passes all these arguments by *reference* (default).

SimRef.f

real*16, SPARC only

```
        subroutine SimRef ( t, c, i, si, r, d, q )
        logical*1  t
        character  c
        integer    i*4, si*2
        real       r*4, d*8, q*16
        t = .true.
        c  = 'z'
        i  = 9
        si = 9
        r  = 9.9
        d  = 9.9
        q = 9.9
        return
        end
```

C passes the *address* of each.

SimRefmain.c

```
main ( )
{
   char    t ;
   char    c ;
   int     i ;
   short   si ;
   float   r ;
   double  d ;
   long double  q = 5.5 ;
   extern simref_ ( char *t, char *c, int *i, short *si,
          float *r, double *d, long double *q ) ;
   simref_ ( &t, &c, &i, &si, &r, &d, &q ) ;
   printf ( "%08o %c %d  %d %3.1f %3.1f %L3.1f \n",
              t,   c, i, si,    r,    d,    q ) ;
}
```

Simple types passed by reference.

```
demo$ f77 -c -silent SimRef.f
demo$ cc -c SimRefmain.c
demo$ f77 SimRef.o SimRefmain.o    ← This does the linking.
demo$ a.out
00000001 z 9  9 9.9 9.9 9.9
demo$ ■
```

Above, the second `f77` command does the linking.

Complex Types Passed by Reference (C Calls `f77`)

The complex types require a simple structure.

CmplxRef.f

```
      subroutine CmplxRef ( w, z )
      complex w
      double complex z
      w = ( 6, 7 )
      z = ( 8, 9 )
      return
      end
```

Above, `w` and `z` are passed by reference (default).

CmplxRefmain.c

```
main ( )
{
     struct complex { float r, i; };
     struct complex d1;
     struct complex *w = &d1;
     struct dcomplex { double r, i; };
     struct dcomplex d2;
     struct dcomplex *z = &d2;
     extern cmplxref_ ();
     cmplxref_ ( w, z );
     printf ( "%3.1f %3.1f \n%3.1f %3.1f \n",
               w->r, w->i, z->r, z->i );
}
```

Above, `w` and `z` are pointers, so if you pass `w` and `z`, you pass the address. This is pass by reference.

Compile and execute, with output.

```
demo$ f77 -c -silent CmplxRef.f
demo$ cc -c CmplxRefmain.c
demo$ f77 CmplxRef.o CmplxRefmain.o
demo$ a.out
6.0 7.0
8.0 9.0
demo$ ■
```

Character Strings Passed by Reference (C Calls `f77`*)*

Passing strings between C and FORTRAN is not encouraged.

Rules for Passing Strings

- All C strings pass by reference.
- For each FORTRAN argument of character type, an *extra argument* is passed giving the length of the string. The extra argument is equivalent to a C `long int` passed by value. This is nonstandard.
- The order of arguments is as follows:
  - A list of the regular arguments
  - A list of lengths, one for each character argument, as a `long int`
  - The list of extra arguments comes after the list of regular arguments.

Example: Character strings passed by reference. A FORTRAN call.

```
      CHARACTER*7 S
      INTEGER B(3)
…
      CALL SAM( B(2), S )
```

The above call is equivalent to the C call in the following.

```
      char s[7];
      long b[3];
…
      sam_( &b[1], s, 7L );
```

If you make a string in FORTRAN, you must provide an explicit null terminator because FORTRAN does not automatically do that, and C expects it.

Ignoring the Extra Arguments of Passed Strings

You can *ignore* the extra arguments, since they are after the list of other arguments.

The following FORTRAN subroutine gets no values of the extra arguments from the C main.

StrRef.f

```
      subroutine StrRef ( a, s )
      character a*10, s*80
      a = 'abcdefghi' // char(0)
      s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
      return
      end
```

The following C main *ignores* the extra arguments.

StrRefmain.c

```
main ( )
{
     char s10[10], s80[80];
     strref_ ( s10, s80 );
     printf ( " s10='%s' \n s80='%s' \n", s10, s80 );
}
```

Above, C strings pass by reference.

Compile and execute, with output.

```
demo$ f77 -c -silent StrRef.f
demo$ cc -c StrRefmain.c
demo$ f77 StrRef.o StrRefmain.o
demo$ a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
demo$
```

Using the Extra Arguments of Passed Strings

You can *use* the extra arguments.

The following FORTRAN routine *uses* the extra arguments (the sizes) implicitly. The FORTRAN source code cannot use them explicitly.

StrRef2.f

```
      subroutine StrRef2 ( a, s )
      character a*(*), s*(*)
      a = 'abcdefghi' // char(0)
      s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
      return
      end
```

The following C main *passes* the extra arguments explicitly.

StrRef2main.c

```
main ( )
{
     char s10[10], s80[80] ; /*Provide memory for the strings*/
     long  L10, L80 ;
     L10 = 10 ;              /*Initialize extra args*/
     L80 = 80 ;
     strref2_ ( s10, s80, L10, L80 ) ; /*pass extra args to f77*/
     printf ( " s10='%s' \n s80='%s' \n", s10, s80 ) ;
}
```

Above, C strings pass by reference.

Compile and execute, with output.

```
demo$ f77 -c -silent StrRef2.f
demo$ cc -c StrRef2main.c
demo$ f77 StrRef2.o StrRef2main.o
demo$ a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
demo$
```

Arguments Passed by Value (C Calls `f77`) - N/A

FORTRAN can call C, and pass an argument by *value*. But FORTRAN cannot handle an argument passed by *value* if C calls FORTRAN. The work-around is to pass all arguments by *reference*.

Function Return Values (C Calls f77*)*

For function return values, a FORTRAN function of type BYTE, INTEGER, LOGICAL, DOUBLE PRECISION, or REAL*16 (quadruple precision) is equivalent to a C function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

Return an int *(C Calls* f77*)*

Example: FORTRAN returns an INTEGER function value to C.

RetInt.f

```
      integer function RetInt ( k )
      integer k
      RetInt = k + 1
      return
      end
```

RetIntmain.c

```
main()
{
     int k, m;
     extern int retint_ ();
     k = 8;
     m = retint_ ( &k );
     printf( "%d %d\n", k, m );
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent RetInt.f
demo$ cc -c RetIntmain.c
demo$ f77 RetInt.o RetIntmain.o
demo$ a.out
8 9
demo$ ■
```

Return a `float` *(C Calls* `f77`*)*

Example: FORTRAN returns a `REAL` to a C `float`.

RetFloat.f

```
      real function RetReal ( x )
      real x
      RetReal = x + 1.0
      return
      end
```

RetFloatmain.c

```
main ( )
{
        float r, s ;
        extern  float retreal_ () ;
        r = 8.0 ;
        s = retreal_ ( &r ) ;
        printf( " %8.6f %8.6f \n", r, s ) ;
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent RetFloat.f
demo$ cc -c RetFloatmain.c
demo$ f77 RetFloat.o RetFloatmain.o
demo$ a.out
 8.000000 9.000000
demo$ ■
```

In earlier versions of C, if C returned a function value that was a `float`, C promoted it to a `double`, and various tricks were needed to get around that.

Return a `double` *(C Calls* `f77`*)*

Example: FORTRAN returns a `DOUBLE PRECISION` function value to C.

RetDbl.f

```
      double precision function RetDbl ( x )
      double precision x
      RetDbl = x + 1.0
      return
      end
```

RetDblmain.c

```
main()
{
     double x, y;
     extern double retdbl_ ();
     x = 8.0;
     y = retdbl_ ( &x );
     printf( "%8.6f %8.6f\n", x, y );
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent RetDbl.f
demo$ cc -c RetDblmain.c
demo$ f77 RetDbl.o RetDblmain.o
demo$ a.out
8.000000 9.000000
demo$ ■
```

Return a `long double` *(C Calls* `f77`*)*

Example: FORTRAN returns a `REAL*16` to a C `long double`.

RetQuad.f

real*16, SPARC only

```
      real*16 function  RetQuad ( x )
      real*16 x
      RetQuad = x + 1.0
      return
      end
```

RetQuadmain.c

```
main ( )
{
        long double r, s ;
        extern  long double retquad_ ( long double * ) ;
        r = 8.0 ;
        s = retquad_ ( &r ) ;
        printf( " %8.6Lf %8.6Lf \n", r, s ) ;
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent RetQuad.f
demo$ cc -c RetQuadmain.c
demo$ f77 RetQuad.o RetQuadmain.o
demo$ a.out
 8.000000 9.000000
demo$ ■
```

Return a COMPLEX *(C Calls* f77*)*

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is shown below.

```
      COMPLEX FUNCTION F ( … )
```

The pattern for a corresponding C function is as follows.

```
f_( temp, … )
struct { float r, i; } *temp;
```

Example: FORTRAN returns a COMPLEX to a C struct.

RetCmplx.f

```
      complex function RetCmplx ( x )
      complex x
      RetCmplx = x * 2.0
      return
      end
```

RetCmplxmain.c

```
main ( )
{
      struct complex { float r, i; };
      struct complex c1, c2;
      struct complex *w = &c1, *t = &c2;
      extern retcmplx_ ();
      w -> r = 4.0;
      w -> i = 4.5;
      retcmplx_ ( t, w );
      printf ( " %3.1f %3.1f \n %3.1f %3.1f \n",
      w -> r, w -> i, t -> r, t -> i );
}
```

Return a COMPLEX. Compile, link, and execute, with output.

```
demo$ f77 -c -silent RetCmplx.f
demo$ cc -c RetCmplxmain.c
demo$ f77 RetCmplx.o RetCmplxmain.o
demo$ a.out
 4.0 4.5
 8.0 9.0
demo$ ■
```

Using f77 to pass files to the linker tells the linker to use the f77 libraries.

Return a Character String (C Calls f77*)*

Passing strings between C and FORTRAN is not encouraged.

A FORTRAN string function has two extra initial arguments — data address and length.

Example: A FORTRAN function of the following form, with no C() pragma:

```
      CHARACTER*15 FUNCTION G ( … )
```

A C function of the following form:

```
g_ ( result, length, … )
char result[ ];
long length;
```

The above two functions are equivalent, and can be invoked in C as follows:

```
char chars[15];
g_ ( chars, 15L, … );
```

The lengths are passed by value. You must provide the null terminator.

RetChr.f

```
      FUNCTION RetChr( C, N )
      CHARACTER RetChr*(*), C
      RetChr = ''
      DO I = 1, N
           RetChr(I:I) = C
      END DO
      RetChr(N+1:N+1) = CHAR(0) ! Put in the null terminator.
      RETURN
      END
```

Return a character string (*continued*).

RetChrmain.c

```
main()
{ /* Use a FORTRAN 77 character function, (C calls f77) */
        char strbuffer[9] = "123456789" ;
       char *rval_ptr = strbuffer ;      /* extra initial arg 1 */
      int rval_len = sizeof(strbuffer) ; /* extra initial arg 2 */
        extern void retchr_() ;
        char ch = '*' ;
        int n = 4 ;
        int ch_len = sizeof(ch) ;          /* extra final arg */
        printf( " '%s'\n", strbuffer ) ;
        retchr_ ( rval_ptr, rval_len, &ch, &n, ch_len ) ;
        printf( " '%s'\n", strbuffer ) ;
}
```

Compile, link, and execute, with output.

```
demo$ f77 -c -silent RetChr.f
demo$ cc -c RetChrmain.c
demo$ f77 RetChr.o RetChrmain.o
demo$ a.out
 '123456789'
 '****'
demo$ ■
```

The caller must set up more actual arguments than are apparent as formal parameters to the FORTRAN function.

- Arguments that are lengths of character strings are passed by *value.*
- Arguments that are *not* lengths of character strings are passed by *reference.*

Labeled Common (C Calls f77*)*

C and FORTRAN can share values in labeled common. Any of the options that change size or alignment (or any equivalences that change alignment) might invalidate such sharing.

The method is the same no matter which language calls which, but a copy is put in here just to save your flipping back to the other one.

Labeled common.

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common / ilk / u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

UseCommain.c

```
#include <stdio.h>
extern struct comtype {
     float p;
     float q;
     float r;
};
extern struct comtype ilk_;
main()
{
     char *string = "abc0";
     int count = 3;
     extern int usecom_ ( );
     ilk_.p = 1.0;
     ilk_.q = 2.0;
     ilk_.r = 3.0;
     usecom_ ( string, count );
     printf(" ilk_.p=%4.1f, ilk_.q=%4.1f, ilk_.r=%4.1f\n",
     ilk_.p, ilk_.q, ilk_.r );
}
```

Compile and execute, with output.

```
demo$ f77 -c -silent UseCom.f
demo$ cc -c UseCommain.c
demo$ f77 UseCom.o UseCommain.o
demo$ a.out
 ilk_.p = 7.0, ilk_.q = 8.0, ilk_.r = 9.0
demo$ ■
```

Sharing I/O (C Calls `f77`*)*

Mixing FORTRAN I/O with C I/O is not recommended. If you must mix them, it is usually safer to pick one and stick with it, rather than alternating.

The FORTRAN I/O library uses the C standard I/O library. Every open unit in a FORTRAN program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

For sharing I/O, if a C main program calls a FORTRAN subprogram, then there is no automatic initialization of the FORTRAN I/O library (connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively). If a FORTRAN function attempts to reference the `stderr` stream (unit 0), then any output is written to a file named `fort.0` instead of to the `stderr` stream.

Initialize I/O

To make the C program initialize I/O (establish the preconnection of units 0, 5, and 6), do the following:

1. Insert the following line at the start of the C main.

```
    f_init();
```

2. At the end of the C main, insert the following line.

```
    f_exit();
```

The second insertion may not be strictly necessary.

Example: Sharing I/O using a C main and a FORTRAN subroutine.

MixIO.f

```
      subroutine MixIO ( n )
      integer n
      if ( n .LE. 0 ) then
           write(0,*) "error: negative line #"
           n = 1
      end if
      write(*,'("In FORTRAN: line # =",i2)') n
      end
```

MixIOmain.c

```
#include <stdio.h>
main ( )
{
     int i, m = -9;
     f_init();
          for ( i=0; i<=4; i++ ) {
          m++;
          if ( m == 2 || m == 4 )
               printf("In C: line # = %d\n",m);
          else
               mixio_ ( &m );
     }
     f_exit();
}
```

Insertion 1 ⟶

Insertion 2 ⟶

Compile and execute, with output.

```
demo$ f77 -c -silent MixIO.f
demo$ cc -c MixIOmain.c
demo$ f77 MixIO.o MixIOmain.o
demo$ a.out
error: negative line #
In FORTRAN: line # = 1
In C: line # = 2
In FORTRAN: line # = 3
In C: line # = 4
In FORTRAN: line # = 5
demo$ ■
```

With a C main program, the following FORTRAN library routines may not work correctly: `signal()`, `getarg()`, `iargc()`.

13

Alternate Returns (C Calls `f77`*)*

Some C programs need to use a FORTRAN subroutine that has nonstandard returns. To C, such subroutines return an `int` (`INTEGER*4`). The return value specifies which alternate return to use. If the subroutine has no entry points with alternate return arguments, the returned value is undefined.

Example: One regular argument and two alternate returns.

AltRet.f

```
      subroutine AltRet ( i, *, * )
      integer i, k
      i = 9
      k = 20
      if ( k .eq. 10 ) return 1
      if ( k .eq. 20 ) return 2
      return
      end
```

C invokes the subroutine as a function.

AltRetmain.c

```
main()
{
     int k, m ;
     extern  int  altret_ () ;
     k = 0 ;
     m = altret_ ( &k ) ;
     printf( "%d %d\n", k, m ) ;
}
```

Compile, link, and execute.

```
demo$ f77 -c -silent AltRet.f
demo$ acc -c AltRetmain.c
demo$ f77 AltRet.o AltRetmain.o
demo$ a.out
9 2
demo$
```

In this example, the C main receives a 2 as the return value of the subroutine because “`RETURN 2`” was executed.

Runtime Error Messages A

This appendix is organized into the following sections.

Operating System Error Messages	*page 283*
Signal Handler Error Messages	*page 284*
I/O Error Messages	*page 284*

The FORTRAN I/O library, the FORTRAN signal handler, and parts of the operating system (when called by FORTRAN library routines) can all generate FORTRAN error messages.

A.1 Operating System Error Messages

Operating system error messages include system call failures, C library errors, and shell diagnostics. The system call error messages are found in `intro` (2). System calls made through the FORTRAN library do not produce error messages directly. The following system routine in the FORTRAN library calls C library routines which produce an error message.

```
      CALL SYSTEM("rm /")
      END
```

The following message is printed.

```
rm: / directory
```

A

A.2 Signal Handler Error Messages

Before beginning execution of a program, the FORTRAN library sets up a signal handler (`sigdie`) for signals that can cause termination of the program. `sigdie` prints a message that describes the signal, flushes any pending output, and generates a core image, and a traceback.

Presently the only arithmetic exception caught is the `INTEGER*2` division with a denominator of zero. All other arithmetic exceptions are silently ignored.

A signal handler error example follows when the subroutine `SUB` tries to access parameters that are not passed to it:

```
      CALL SUB()
      END
      SUBROUTINE SUB(I,J,K)
      I=J+K
      RETURN
      END
```

The following error message is printed:

```
*** Segmentation violation
Illegal instruction (core dumped)
```

A.3 I/O Error Messages

The following error messages are generated by the FORTRAN I/O library. The error numbers are returned in the `IOSTAT` variable if the `ERR` return is taken.

As an example, the following program tries to do an unformatted write to a file opened for formatted output:

```
WRITE( 6 ) 1
END
```

and gets an error messages like the following:

```
sue: [1003] unformatted io not allowed
logical unit 6, named 'stdout'
lately: writing sequential unformatted external IO
Illegal instruction (core dumped)
```

The following error messages are generated. These same error messages appear online at the end of the `man` page `perror`(3f).

If the error number is less than 1000, then it is a *system* error. See `intro` (2).

1000 `error in format`

Read the error message output for the location of the error in the format. It can be caused by more than 10 levels of nested parentheses or an extremely long format statement.

1001 `illegal unit number`

It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is 2^{31} - 1.

1002 `formatted io not allowed`

The logical unit was opened for unformatted I/O.

1003 `unformatted io not allowed`

The logical unit was opened for formatted I/O.

1004 `direct io not allowed`

The logical unit was opened for sequential access, or the logical record length was specified as 0.

1005 `sequential io not allowed`

The logical unit was opened for direct access I/O.

1006 `can't backspace file`

You cannot do a seek on the file associated with the logical unit, and therefore you cannot do a backspace. The file may be a `tty` device or a pipe.

1007 `off beginning of record`

You tried to do a left tab to a position before the beginning of an internal input record.

1008 `can't stat file`

The system can't return status information about the file. Perhaps the directory is unreadable.

1009 `no * after repeat count`

Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

1010 `off end of record`

A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

1011 *<not used>*

1012 `incomprehensible list input`

List input has to be as specified in the declaration.

1013 `out of free space`

The library dynamically creates buffers for internal use. You ran out of memory for this (that is, your program is too big).

1014 `unit not connected`

The logical unit was not open.

1015 `read unexpected character`

Certain format conversions can't tolerate nonnumeric data.

1016 `illegal logical input field`

`logical` data must be T or F

1017 `'new' file exists`

You tried to open an existing file with `status='new'`.

1018 `can't find 'old' file`

You tried to open a nonexistent file with `status='old'`.

1019 `unknown system error`

Shouldn't happen, but ...

1020 `requires seek ability`

You tried to do a seek on a file that doesn't allow that. Some of the ways of doing I/O that require the ability to do a seek are:
(1) Direct access, (2) Sequential unformatted I/O, (3) Tabbing left.

1021 `illegal argument`

Certain arguments to `open` (and related functions) are checked for legitimacy. Often only nondefault forms are checked.

1022 `negative repeat count`

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

1023 `illegal operation for unit`

You tried to do an I/O operation that is not possible for the device associated with the logical unit. You get this error if you try to read past end-of-tape, or end-of-file.

1024 *<not used>*

1025 `incompatible specifiers in open`

You tried to open a file with the `'new'` option and the `access='append'` option, or some other invalid combination.

1026 `illegal input for namelist`

A namelist read encountered an invalid data item.

1027 `error in FILEOPT parameter`

Using `OPEN`, the `FILEOPT` string has bad syntax.

For example, the following kind of error message is printed:

```
open: [1027] error in FILEOPT parameter
logical unit 8, named 'temp'
Abort
```

A

XView Toolkit

B

This appendix is organized into the following sections.

XView Overview	*page 289*
FORTRAN Interface	*page 290*
C to FORTRAN	*page 297*
Sample Programs	*page 299*
References	*page 301*

FORTRAN 3.0.1 and OpenWindows 3.x

This appendix introduces the FORTRAN interface to the XView programmer's toolkit. It is assumed that you are familiar with the XView windows system from the *user* point of view — that is, you know the appearance and function of the windows, scrollbars, menus, and so forth. It is also assumed that you are familiar with XView from a *programmer's* point of view, as described in the *XView Programming Manual*.

B.1 XView Overview

The XView Application Programmer's Interface is an object-oriented, server-based, user-interface toolkit for the X Window System Version 11 (X11). [1] It is designed and documented to help C programmers manipulate XView windows and other XView objects.

1. The X Window System is a product of the Massachusetts Institute of Technology.

B

Tools

This kit is a collection of functions. The runtime system is based on each application having access to a server-based *Notifier*, which distributes input to the appropriate window, and a *window manager* which manages overlapping windows. There is also a *Selection Service* for exchanging data between windows (in the same or different processes).

Objects

XView is an *object-oriented* system. XView applications create and manipulate XView objects. All such objects are associated with XView packages. Objects in the same package share common properties. The major objects are windows, icons, cursors, menus, scrollbars, and frames. A *frame* contains non-overlapping subwindows within its borders. You manipulate an object by passing a unique identifier or *handle* for that object to procedures associated with that object.

B.2 FORTRAN Interface

This chapter focuses on manipulating XView windows and objects with FORTRAN. The FORTRAN XView interface consists of a set of header files and an interface library. To write XView applications, you need to use the library, header files, object handles, and standard procedures.

Compiling

The library `Fxview` provides a FORTRAN interface to XView. The actual XView procedures are in the libraries `xview` and `X11`. To compile an XView program, you include some header files and link the libraries.

Example: *Solaris 2.x*.

If you use `pixrect`, you must put in `-lpixrect` before `-xvol`.

```
demo$ f77 -U -Nx2000 -Nn4000 -o app \
 -I/opt/SUNWspro/SC3.0.1/include/f77 app.F \
 -lFxview -lxview -lolgx -lX11
```

Initializing

Initialize the Xview library using the `xv_init` function. Some of the functions require special initialization and some do not, so in general it is safer to do this initialization. There are special aspects about this initialization.

- This initialization must be done before the FORTRAN main program starts executing its internal initialization code.
- There is a special function named `f77_init` that each FORTRAN main program always calls before executing its internal initialization code. The version of `f77_init` provided in `libF77` does nothing, but you can provide a substitute version to do the initialization.

The following example shows one way to use `f77_init` to invoke `xv_init`. Example: Use `f77_init` to invoke `xv_init`.

```
demo$ cat xvini.c
#include <xview/xview.h>
void
f77_init(int *argcp, char ***argvp, char ***envp)
{
    xv_init(XV_INIT_ARGC_PTR_ARGV, argcp, *argvp) ;
    f77_no_handlers = 1 ;                  /* See paragraph below. */
}
demo$ cc -o xvini xvini.c
demo$ f77 xvini.o Any.f
```

Signal Handlers and `f77_no_handlers`

The global variable `f77_no_handlers` is a flag that affects subsequent initialization routines. If it is nonzero, the FORTRAN runtime system does not set up any signal handlers.

Signal handlers are for dealing with floating-point exceptions, interrupts, bus errors, segmentation violations, illegal instructions, and so forth.

The specific problem with XView is that many XView programs do their own signal handling. These programs fail if the FORTRAN runtime system sets and uses the normal signal handlers. These normal signal handlers intercept signals, flush the output buffers and print a descriptive message. If you have two sets of signal handlers in the same program, they interfere with each other.

Example: *Solaris 1.x*.

```
demo$ f77 -U -Nx2000 -Nn4000 -o app \
 -I/usr/lang/SC3.0.1/include/f77 app.F \
 -lFxview -lxview -lolgx -lX11
```

Header Files

The header files define the necessary data types, constants, and external procedures necessary to write programs using the XView interface with FORTRAN.

Names of Header Files

Every XView program needs the header file `stddefs_F.h` for standard definitions. It must be first.

The names of the header file are the same as the XView C language header files with the `.h` extension changed to the `_F.h` extension. For example, the FORTRAN header file corresponding to the XView file `panel.h` is named `panel_F.h`. Other header files are `canvas_F.h`, `text_F.h`, and so forth.

In addition to the header files corresponding to the XView headers, there are three additional ones. They are

- `stddefs_F.h`

  This defines some basic types that are used by most of the other FORTRAN XView header files. You must include this file before any other of the FORTRAN XView files.

- `undef_F.h`

  This is used if you have more than one subroutine in a single file that needs the XView data types. This file undefines certain symbols which are used in the header files so that you can include the header files in multiple subroutines or functions in the same source file.

- `procitf_F.h`

  This header file contains declarations for routines which will generate interface routines for all procedure types which are passed to XView. Some of the features of XView require you to

provide a subroutine that is called by XView when certain events happen. Since FORTRAN routines pass arguments differently than C routines, and since XView assumes the C calling conventions, interface routines are needed to map the arguments correctly. The input argument is a FORTRAN subroutine. The output is the address of an interface routine that will call the FORTRAN routine with the arguments properly mapped.

Example: Interpose event call.

```
      EXTERNAL event_func, my_repaint
      CALL set_CANVAS_REPAINT_PROC ( canvas,
&          canvas_repaint_itf ( my_repaint ) )
      err = notify_interpose_event_func ( frame,
&          notify_event_itf ( event_func ),
&          NOTIFY_IMMEDIATE )
```

There can be at most 30 different interface procedures of each type.

Usage of Header Files

To use header files with FORTRAN, do three things.

- Specify the option `-I/opt/SUNWspro/include/f77` in the compile command.
- In your source file put a line

```
#include "stddefs_F.h"
```

Put in such `include` lines for any other header files you need.

- Make `.F` the suffix of your source file.

When you compile a FORTRAN source file that has a `.F` suffix, the C preprocessor will replace the `#include` line with the contents of the specified file.

Generic Procedures

There is one general initialization procedure `xv_init()`.

These are the standard generic procedures for things you do to objects.

- `xv_create()`

- xv_find()
- `xv_destroy()`

However, some special procedures for FORTRAN are available. See “Attribute Lists” for details.

Attribute Procedures

Each class of objects has its own set of *attributes*. Each attribute has a predefined, or default, value. For example, for the class of scrollbars, there is a width and a color.

The standard C interface to XView defines two routines `xv_get()` and `xv_set()`, which get and set attributes of XView objects. These routines take an arbitrary number and type of parameters and return various types depending on its arguments.

Instead of these routines, the FORTRAN interface to XView defines a separate routine to get and set each attribute.

set — The routine to set an attribute is named `set_attrname`

- Each set routine is a subroutine.
- Each set routine takes, as its first argument, the object for which the attribute is being set.
- The second argument is the value of the attribute.

get — The routine to get the value of an attribute is named `get_attrname`

- Each set routine is a function.
- Each get routine takes an XView object as the first argument
- Each get routine returns the value of the attribute requested.

For example

```
      CALL set_WIN_SHOW ( frame, TRUE )
      width = get_CANVAS_WIDTH ( canvas )
```

Attribute Lists

Some of the XView C routines may optionally take extra arguments that are lists of attributes and values. The extra arguments vary in number and type.

The FORTRAN versions of these routines do *not* support this variable number of arguments, and these versions ignore any arguments after the required ones. However a 0 must be passed as the last argument to be compatible with future versions which may support the extra arguments.

Instead, special versions of these routines are provided that take as a last argument an argument of type `Attr_avlist`. This type is a pointer to an array of attributes and values. The special routines are:

- `xv_init_l()`
- `xv_create_l()`
- `xv_find_l()`
- `selection_ask_l()`
- `selection_init_request_l()`

Example calls:

`mymenu` is an object of type `XV_object`.

```
      mymenu = xv_create ( NULL, MENU, 0 )
      ncols = get_MENU_NCOLS ( mymenu )
      call set_MENU_NITEMS ( mymenu, items )
      call xv_find_l ( mymenu, MENU,
&          attr_create_list_2s ( MENU_DEFAULT,4) )
      call xv_destroy( mymenu )
```

The lists for `Attr_avlist` are created by functions which have the names

- `attr_create_list_n()`
- `attr_create_list_ns()`

- The *n* indicates the number of arguments the routine accepts.
- The number of arguments can be 1-16.
- The routines ending in `s` return a pointer to a static attribute-value array which is reused with each call to the static list routines.
- The versions without the `s` return a dynamically allocated list which you pass to `xv_destroy()` when you are finished with the list.
- For any attribute which expects a pointer, you need to pass `loc()` of the address of the object, instead of the address of the object. You need to do this because these routines know that FORTRAN passes arguments by reference, so they always dereference each argument. This usage is shown in the last sample in this appendix.

B

Handles

If you create an XView object, then `xv_create()` returns a *handle* for that object. You pass this handle to the appropriate procedure for manipulating the object.

Data Types

Each XView object has its own specific data type. The name of an object's data type always starts with a capital letter. For example, the data type for a scrollbar is `Scrollbar`. The standard list of these types is in the header files.

Coding Fragment

This sample is provided to illustrate the style of programming with the XView interface in FORTRAN. It does three things.

- Creates a scrollbar with a page length of 100 units and a starting offset of 10
- Changes the page length to 20
- Destroys the scrollbar

```
      Scrollbar bar
      bar = xv_create_l ( 0, SCROLLBAR,
&          attr_create_list_4s ( SCROLLBAR_PAGE_LENGTH, 100,
&                                SCROLLBAR_VIEW_START, 10 )
&           )
      call set_SCROLLBAR_PAGE_LENGTH ( bar,  20 )
      call xv_destroy ( bar )
```

Note the following about this example.

- `bar` is declared to be of type `Scrollbar`.
- `xv_create_l()` is invoked as a *function*.
- `set_SCROLLBAR_PAGE_LENGTH()` is invoked as a *subroutine*.
- `xv_destroy()` is invoked as a *subroutine*.

B.3 C to FORTRAN

The Problem — Besides the six standard generic procedures, there are approximately 80 other procedures, plus hundreds of attributes. These are all documented in the manual *XView 1.0 Reference Manual: Summary of the XView API*. The problem is that all of the coding is in C.

The Easy Part — You can use the following items of information as you find them in the manual, with no change.

- XView procedure names
- XView object names
- XView object data types (except `Boolean`, more on this below)

The Hard Part — You must make the following changes.

- Any elementary C data type used must be converted to the corresponding FORTRAN data type.
- If a C procedure *returns* something, then it must be invoked in FORTRAN as a *function*; otherwise it must be invoked as a *subroutine.*
- The XView type `Boolean` must be converted to the FORTRAN type `LOGICAL`.
- Arguments which are declared as *type*`*` have a FORTRAN type of *type*`_ptr`.
- Arguments which have a type of `struct` *str* have a FORTRAN type of `Str`.

This table summarizes converting C declarations to FORTRAN.

Table B-1 Matching C and FORTRAN Declarations

C	FORTRAN
`short int x;`	`INTEGER*2 X`
`long int x;`	`INTEGER*4 X`
`int x;`	`INTEGER*4 X`
`char x;`	`BYTE X or LOGICAL*1 X`
`char *x;`	`CHARACTER*n X` *{Read Note}*
`char x[6];`	`CHARACTER*6 X`
`float x;`	`REAL X`
`double x;`	`DOUBLE PRECISION X`

In standard FORTRAN, variables of type `INTEGER`, `LOGICAL`, and `REAL` use the same amount of memory. Since `LOGICAL*1` or `BYTE` violate such rules, they are not standard FORTRAN and can result in nonportable programs.

Note – The C declaration for "*x is a pointer to a character*" is "`char*x`." The FORTRAN declaration for "*X is a character string*" is "`CHARACTER*n X`," where *n* can be any size; but the FORTRAN character string itself must be null-terminated. These two declarations are equivalent.

If you use a character constant, you get it null-terminated automatically. If you use a character variable, you have to terminate it explicitly with a null character, `CHAR(0)`.

Example: Terminate a variable character string with the null character.

```
CHARACTER X*10, Z*20
X = 'abc'
Z = X // CHAR(0)
```

Sample Translation: C Function Returning Something

In the manual *XView 1.0 Reference Manual: Summary of the XView API*, in the chapter "XView Procedures and Macros," you find this entry:

```
texsw_insert()
      Inserts characters in buf into textsw at the current insertion
      point. The number of characters actually inserted is returned.
      This will equal buf_len unless there was a memory allocation
      failure. If there was a failure, it will return 0.
      Textsw_index
      textsw_insert(textsw, buf, buf_len)
      Textsw textsw;
      char buf;
      int buf_len;
```

Translation to FORTRAN:

- Leave the object data type `Textsw` as is.
- Since it returns the number of characters inserted, invoke it as a function.

```
Textsw textsw
CHARACTER*4 buf
INTEGER*4 N, buf_len, textsw_insert
…
buf(4:4) = CHAR(0)
N = textsw_insert(textsw, buf, buf_len)
IF ( N .EQ. 0 ) …
```

Sample Translation: C Function Returning Nothing

In the same manual and chapter, we find this entry.

```
frame_set_rect()
      sets the rect of the frame. X, y is the upper left corner of the
      window coordinate space. Width and height include the window  decoration.
            void
            frame_set_rect(frame,rect)
            Frameframe;
            Rectrect;
```

Translation to FORTRAN: It does not return anything, invoke as a subroutine.

```
      …
      Frame frame
      Rect  rect
      …
      CALL frame_set_rect ( frame, rect )
      …
```

B.4 Sample Programs

Some of the XView C routines (such as `xv_create()`) take a variable number of arguments. The corresponding FORTRAN versions do not. They ignore any arguments passed after the required arguments.

Alternate versions of the variable-argument-list routines are provided with an "`_l`"appended to the name. The final argument is an attribute-value list which can be created by the `attr_create_list_*` routines.

Sample 1: Hello World. A small FORTRAN program using the XView toolkit.

```
demo$ cat  xhello.F
      PROGRAM hello1F
#include "stddefs_F.h"
#include "frame_F.h"
#include "panel_F.h"
#include "window_F.h"
#include "attrgetset_F.h"
      EXTERNAL loc
      Frame base_frame                        ! Three special
      Panel panel                             ! XView
      Xv_panel_or_item_ptr pi                 ! type statements
      base_frame = xv_create ( 0, FRAME, 0 )
      panel = xv_create_l ( base_frame, PANEL, 0 )
      pi = xv_create_l ( panel, PANEL_MESSAGE,
&          attr_create_list_2s ( PANEL_LABEL_STRING,
&                                loc("Hello world!"))
&          )
      CALL window_main_loop ( base_frame )
      END
demo$ ■
```

Compile.

Example: *Solaris 2.x*.

```
demo$ f77 -U -Nn5000 -Nx2000 -o hello_world \
          xhello.F -lFxview \
           -I/opt/SUNWspro/SC3.0.1/include/f77 \
           -lxview -lolgx -lX11
xhello.F:
     MAIN xhello:
demo$ ■
```

Example: *Solaris 1.x*.

```
demo$ f77 -U -Nn5000 -Nx2000 -o hello_world \
          xhello.F -lFxview \
           -I/usr/lang/SC3.0.1/include/f77 \
           -lxview -lolgx -lX11
xhello.F:
     MAIN xhello:
demo$ ■
```

You will get many warning messages about names being over 32 characters. To suppress these messages, include a `-w` option.

Run the executable file.

```
demo$ hello_world
```

Soon after you run the executable file, the window will come up as a single frame, with the words "`hello world`" in the frame header.

Sample 2: Create a `tty` subwindow and run `/bin/ls` in it.

```
#include "stddefs_F.h"
#include <textsw_F.h>
#include <frame_F.h>
#include <panel_F.h>
#include <termsw_F.h>
#include <tty_F.h>
        Frame           frame
        character*8     command
        Termsw          tty
        integer         my_argv(2)
        command = '/bin/ls' // char(0)
        my_argv(1) = loc(command)
        my_argv(2) = 0
        call xv_init(0)
        frame = xv_create(0, FRAME, 0)
        tty = xv_create_l(frame, TERMSW,
     1          attr_create_list_2(TTY_ARGV, loc(my_argv)), 0)
        call set_TERMSW_MODE(tty, TTYSW_MODE_TYPE)
        call set_WIN_ROWS(tty, 24)
        call set_WIN_COLUMNS(tty, 80)
        call window_fit(frame)
        call window_main_loop( frame )
        end
```

Note that we pass `loc(my_argv)` to `attr_create_list_2()`.

B.5 References

A comprehensive programmer's reference manual for XView is now available from O'Reilly & Associates, Incorporated, as Volume Seven of their series of *X Window System* documentation.

To order directly, contact O'Reilly & Associates, Incorporated.

XView Programming Manual.

O'Reilly & Associates, Inc. 632 Petaluma Avenue Sebastopol, CA 95472

(800) 338-6887

EMAIL: uunet!ora!xview

B

iMPact: Multiple Processors C

This appendix is organized into the following sections.

Requirements	*page 303*
Overview	*page 304*
Speed Gained or Lost	*page 306*
Number of Processors	*page 307*
Debugging Tips and Hints for Parallelized Code	*page 309*

This appendix introduces ways to spread a set of programming instructions over a multiple-processor system so they execute in parallel. The process is called *parallelizing*. The goal is speed. It is assumed that you are familiar with the concepts of parallel processing and you are familiar with Sun FORTRAN and the SunOS or UNIX operating system.

iMPact multiprocessor FORTRAN includes *automatic* and *explicit* loop parallelization, which are described in the “Overview.”

C.1 Requirements

Multiprocessor FORTRAN requires the following.

- A Sun multiple processor system (SPARCstation-10, 600MP series server, …)
- Solaris 2.2 Operating Environment, or later (has multi-threading)
- iMPact FORTRAN MP

C

The SPARCstation-10 (or a SPARC Server in the 600MP series) can have more than one processor. Solaris 2.2 includes the SunOS 5.2 operating system, which supports the `libthread` library and running many processors simultaneously The SunOS 4.x system does not support `libthread`. FORTRAN MP has features that exploit multiple processors using the SunOS 5.2 operating system.

C.2 Overview

In general, this compiler can parallelize certain kinds of loops that include arrays. You can let the *compiler* determine which loops to parallelize (*automatic* parallelizing) or *you* can specify each loop yourself *(explicit* parallelizing). The parallelizer is integrated tightly with optimization and operates on the same intermediate representation used by the optimizer.

Automatic Parallelization

Automatic parallelization is both fast and safe. To *automatically* parallelize loops, use the `-autopar` option. With this option, the *software* determines which loops are appropriate to parallelize.

Example: *Automatic* parallelization. Do all appropriate loops.

```
demo$ f77 -autopar any.f
```

Explicit Parallelizing

Explicit parallelization may get extra performance and risk of incorrect results. To *explicitly* parallelize all user-specified loops, do the following.

- Determine which loops are appropriate to parallelize.
- Insert a special directive *just before* each loop that you want to parallelize.
- Use the `-explicitpar` option on the compile command line.

The `libthread` Primitives

If you do your own multi-threaded coding using the `libthread` primitives, do *not* use `-autopar` or `-explicitpar`. Either do it all yourself or let the compiler do it. Conflicts and unexpected results may happen if you and the compiler are both trying to manage threads with the same primitives. See *–mt* in Chapter 3, “Using the Compiler.”

Example: *Explicit* parallelization. Do only the "`DO I=1, N`" loop.

```
demo$ cat t1.f
      ...
c$par doall
      do i = 1, n          ! This loop gets parallelized.
           a(i) = b(i) * c(i)
      end do

      do k = 1, m          ! This loop does not get parallelized.
           x(k) = y(k) * z(k)
      end do

      ...
demo$ f77 -explicitpar t1.f
```

The "`c$par doall`" is explained later.

Summary

The following table summarizes the parallel options and the pragma.

Table C-1 Parallelization Summary

Options	**Syntax**	**Risk of Incorrect Results**
Automatic (*only*)	`-autopar`	Normal
Automatic-with-reduction	`-autopar -reduction`	Roundoff
Explicit (*only*)	`-explicitpar`	Note pragma, below.
Automatic and Explicit	`-parallel`	Note pragma, below.
Automatic-with-reduction, and Explicit	`-parallel -reduction`	Note pragma, below.
Show which loops are parallelized	`-loopinfo`	Not Applicable
Show warnings with explicit	`-vpara`	Not Applicable
Pragma		
`doall`	`c$par doall`	High 

Notes on the Parallel Options and the Pragma

- `-reduction` requires `-autopar`.
- `-autopar` includes dependence analysis and loop structure optimization.
- `-parallel` is equivalent to `-autopar -explicitpar`.
- `-explicit -depend` is equivalent to `-parallel`.
- Negations: `-noautopar`, `-noexplicitpar`, `-noreduction`
- The parallelization options can be in any order but must be all lower case.
- All require the iMPact FORTRAN MP enhancement package and Solaris 2.2.
- To get faster code, all require a multiprocessor system; on a single-processor system the code usually runs slower.
- Using `-explicitpar` or `-parallel` has high risk as soon as you insert a pragma.
- You can combine automatic and explicit with the `-parallel` option.
  - The compiler automatically parallelizes all appropriate loops.
  - It also parallelizes any parallelizable loops that you explicitly identify by a pragma (still a risk with pragmas of producing incorrect results).
  - A loop with an explicit pragma gets no reductions.

Standards

Multiprocessing is an evolving concept. When standards for multiprocessing are established, the above features may be superceded.

C.3 Speed Gained or Lost

The speed gained varies widely with the application. Some programs are inherently parallel and show great speedup. Many have no parallel potential and show no speedup. There is such a wide range of improvement that it is hard to predict what speedup any one program will get.

Variations in Speedups

To illustrate the range of possible speedups, the following hypothetical scenario is presented.

Assume 4 Processors

With parallelization the following variations occur. The normal upper limit (with 4 processors) is about *3 times as fast*.

- Many perfectly good programs, tuned for single-processor computation, and with the overhead of the parallelization, *actually run slower.*
- Many perfectly good programs (tuned for single-processor computation) get *absolutely no speedup*.
- Some programs run *10% faster*
- A few less run *50% faster*
- Even fewer run *100% faster*
- A few have so much parallelism that they run 3 or 4 times faster.

Vectorization Comparison

If you have good speedup on vector machines (with an autovectorizing compiler) a first-order rough approximation may be performed as follows.

speedup = vectorization * (number of CPUS -1)

Remember that this is only a first-order rough approximation.

C.4 Number of Processors

To set the number of processors, set the environment variable PARALLEL.

Setting environment variables varies with the shell, csh(1) or sh(1).

Example: Set PARALLEL to 4.

- sh:

```
demo$ PARALLEL=4
demo$ export PARALLEL
```

- csh:

```
demo% setenv PARALLEL  4
```

Guidelines for Number of Processors

The following are general guidelines, not hard and fast rules. It usually helps to be flexible and experimental with number of processors.

For these guidelines, let N be the number of processors on the machine.

- Do *not* set PARALLEL greater than N (can degrade performance seriously)
- Try PARALLEL set to the number of processors wanted *and expected to get*.
- In general, allow at least one processor for activities other than the program you are (for overhead, other users, and so forth).
- For a *one-user*, multiprocessor system, try PARALLEL=N-1 and try PARALLEL=N.
- For a *one-user* system, if the user asks for more processors than are available on the machine, there can be serious degradation of performance.
- For a *multiple-user* system, if all users together ask for more processors than are available on the machine, there can be serious degradation of performance.

  If the machine is overloaded with users it may help to try PARALLEL set to much less than N. For example, with a 10-user machine, it may help to try PARALLEL at 4, or 6, or 8. If you ask for 10 and cannot get 10, then you may end up time-sharing some CPU's with other users.

C.5 Debugging Tips and Hints for Parallelized Code

The debug option conflicts with the parallelizing options.

The Problem

- If you compile a routine with `-g` and a parallelizing option, the compiler turns off the parallelizing option. The `-autopar`, `-explicitpar`, and `-parallel` options conflict with `-g` because `dbx` cannot do useful debugging on a routine compiled with `-parallel`, `-autopar`, or `-explicitpar`.

  Example: With `-g`, `f77` turns off `-parallel`.

```
demo$ f77 -O3 -g -parallel any.f
f77: Warning: -parallel conflicts with -g. -parallel turned off.
demo$
```

See page 312 for a work around.

Some Solutions

Debugging parallelized programs turns out to be a craft, rather than a science. The following tips and hints provide a few things to try for debugging a parallelized program.

- Try it without any parallelization

  You can turn off the parallelization options or set the CPUs to one.
  - Try it without any parallel options.

    Compile and run the program first with `-O3` or `-O4`, but without `-autopar`, `-explicitpar`, and `-parallel` to verify that it works correctly.
  - Try it with only one CPU.

    Run the program with the environment variable `PARALLEL=1`.

  If the problem goes away, then you know it is due to parallelization.

  If the problem does not go away, and you are using `-autopar`, then the compiler is parallelizing something it should not. Some differences may exist because parallelized programs are always optimized.

- Try it without reduction options

  If you are using the `-reduction` option, summation reduction may be occurring and yielding slightly different answers. Try running without this option if you have it turned on.

- Try to reduce the number of compile options

  Try to reduce the number of compile options to the minimum set of `-parallel -O3` and see if you get the correct results.

- Try `fsplit`

  If you have a lot of subroutines in your program, use `fsplit` to break them into separate files. Then compile some with and without `-parallel`. Then use `ld` to link the `.o` files, you will need to use `-parallel` on the `ld` command.

  Execute the binary and verify results.

  Repeat this process until the problem is narowed down do one subroutine.

  You can proceed with #6 or #7 to track down the loop causing the problem.

- Try `-loopinfo`

  Use the `-loopinfo` option to see which loops are being parallelized and which loops are not.

- Try a dummy subroutine

  Create a dummy subroutine or function which does nothing. Put calls to this subroutine in a few of the loops which are being parallelized. Recompile and execute. (Use #5 to get the loops which are being parallelized.)

  Continue this process until you start getting the correct results.

  Then remove the calls from the other loops, compile and execute to verify you are getting the correct results.

- Try `c$par doall` and `-explicitpar`

  Add the `c$par doall` directive to a couple of the loops which are being parallelized. Compile with `-explicitpar`, then execute and see if you are getting the correct results. (Use #5 to get the loops which are being parallelized.) This permits adding I/O statements to the parallelized loop.

  Repeat this process until you find the loop causing the wrong results.

Note – If you need `-explicitpar` only (without `-autopar`), do *not* compile with `-explicitpar` and `-depend`. This is the same as compiling with `-parallel`, which of course includes `-autopar`.

- Try running loops *backwards* serially.

  Replace
  ```
  DO I=1,N
  ```
  with
  ```
  DO I=N,1,-1
  ```

  Different results point to data dependences.

- Avoid using the loop index.

  It is safer to avoid using the loop index in the loop body, especially if the index is used as an argument in a call.

  Replace
  ```
  DO I=1,N
  ```
  with
  ```
  DO I1=1,N
  I=I1
  ```

- Use dbx on a parallel loop

  To use dbx on a parallel loop, temporarily rewrite the program as follows:
  - Isolate the body of the loop in a file and subroutine of its own
  - In the original routine, replace loop body with a call to the new subroutine
  - Compile new subroutine with -g and no parallelization options
  - Compile changed original routine with parallelization and no -g

  Example: Manually transform a loop to allow using dbx in parallel.

Original

We will split loop.f into 2 parts:
Part 1 on loop1.f
Part 2 on loop2.f

Part 1
Loop replaced loop body (the "main")

Part 2
Body of the loop →

Compile Part 1: parallel, no dbx.
Compile Part 2: dbx, no parallel.
Bind both into a.out.
Start a.out under dbx control.

Put a break point into loop body.

Run

dbx stops at the break point.

Show k
Read "***Debugging a Program***", Chapter 12, "Process/Thread Inspector."

```
demo$ cat loop.f
c$par doall
      DO i = 1,10
           WRITE(0,*) 'Iteration ', i
      END DO
      END

demo$ cat loop1.f
c$par doall
      DO i = 1,10
           k = i
           CALL loop_body ( k )
      END DO
      END

demo$ cat loop2.f
      SUBROUTINE loop_body ( k )
      WRITE(0,*) 'Iteration ', k
      RETURN
      END
demo$ f77 -O3 -c -explicitpar loop1.f
demo$ f77 -c -g loop2.f
demo$ f77 loop1.o loop2.o -explicitpar
demo$ dbx a.out                          ← Various dbx messages not shown
(dbx) stop in loop_body
(2) stop in loop_body
(dbx) run
Running: a.out
(process id 28163)
t@1 (l@1) stopped in loop_body at line 2 in file "loop2.f"
    2           write(0,*) 'Iteration ', k
(dbx) print k
k = 1                                    ← Various values other than 1 are possible
(dbx) ■
```

iMPact: Automatic Parallelization D

This appendix is organized into the following sections.

What You Do	*page 313*
What the Compiler Does	*page 314*
Definition: Automatic Parallelizing	*page 316*
Reduction for Automatic Parallelizing	*page 320*

This appendix shows an easy way to parallelize programs for multiple processors. This is called *automatic parallelizing*. This is a "*how to do it*" guide.

D.1 What You Do

To tell the compiler to parallelize *automatically*, use the `-autopar` option. Example: Parallelize automatically, some loops get parallelized, some do not.

See Appendix C, "iMPact: Multiple Processors" for required background.

```
demo$ cat t2.f
      ...
      do i = 1, 1000                ! ⟵ Parallelized
           a(i) = b(i) * c(i)
      end do

      do k = 3, 1000                ! ⟵ Not parallelized -- dependency
           x(k) = x(k-1) * x(k-2) ! See page 316.
      end do
      ...
demo$ f77 -autopar t2.f
```

To determine which programs benefit from automatic parallelization, study the rules the compiler uses to detect parallelizable constructs. Alternatively, compile the programs with automatic parallelization then time the executions.

No Primitives

If you do your own multi-threaded coding using the `libthread` primitives, do *not* use `-autopar`. Either do it all yourself or let the compiler do it. Conflicts and unexpected results may happen if you and the compiler are both trying to manage threads with the same primitives.

See *–mt* in Chapter 3, “Using the Compiler.”

D.2 What the Compiler Does

For *automatic* parallelization, the compiler does two things:

- Dependency analysis to detect loops that are parallelizable
- Parallelization of those loops

This is similar to the analysis and transformations of a vectorizing compiler.

Parallelize the Loop

The compiler applies appropriate dependence-based restructuring transformations. It then distributes the work evenly over the available processors. Each processor executes a different chunk of iterations.

Example: 4 processors, 1000 iterations; the following occur simultaneously.

Processor 1 executing iterations	1	through	250
Processor 2 executing iterations	251	through	500
Processor 3 executing iterations	501	through	750
Processor 4 executing iterations	751	through	1000

Dependency Analysis

A set of operations can be executed in parallel only if the computational result does not depend on the order of execution. The compiler does a dependency analysis to detect loops with no order-dependence. If it errs, it does so on the side of caution. Also, it may not parallelize a loop that could be parallelized because the gain in performance does not justify the overhead.

Example: *Automatic* parallelizing skips this loop; it has data dependencies.

```
do k = 3, 1000
     x(k) = x(k-1) * x(k-2)
end do
```

You cannot calculate `x(k)` until two previous elements are ready.

Definitions: Array, Scalar, and Pure Scalar

- An *array* variable is one that is declared with dimensioning in a `DIMENSION` statement or a type statement (examples below).
- A *scalar* variable is a variable that is not an array variable.
- A *pure scalar* variable is a scalar variable that is not aliased (not referenced in an `equivalence` statement and not in a `pointer` statement).

Examples: Array / scalar, both `m` and `a` are *array* variables; `s` is *pure scalar.*

```
dimension a(10)
real m(100,10), s, u, x, z
equivalence ( u, z )
pointer ( px, x )
s = 0.0
...
```

The variables `u`, `x`, `z`, and `px` are *scalar* variables, but *not pure scalar.*

D

D.3 Definition: Automatic Parallelizing

General Definition

Automatic parallelization parallelizes `DO` loops that have no inter-iteration data dependencies.

Details

This compiler finds and parallelizes any loop that meets the following criteria (but note exceptions below).

- The construct is a `DO` loop (uses the `DO` statement, but not `DO WHILE`).
- The values of *array* variables for each iteration of the loop do not depend on the values of *array* variables for any other iteration of the loop.
- Calculations within the loop do not *conditionally* change any *pure scalar* variable that is referenced after the loop terminates.
- Calculations within the loop do not change a *scalar* variable across iterations. This is called *loop-carried dependency.*

There are slight differences from vendor to vendor, since no two vendors have compilers with precisely the same criteria.

Example: Using the `-autopar` option.

```
      ...
      do i = 1, n                    ! ⟵ Parallelized
           a(i) = b(i) * c(i)
      end do
      ...
demo$ f77 -autopar t.f
```

Apparent Dependencies

Sometimes the dependencies are only apparent and can be eliminated automatically by the compiler. One of the many transformations the compiler does is make and use its own private versions of some of the arrays. Typically, it can do this if it can determine that such arrays are used in the original loops only as temporary storage.

Example: Using `-autopar`, some dependencies eliminated by *private arrays*.

```
      parameter (n=1000)
      real a(n), b(n), c(n,n)
      do i = 1, 1000          ! ← Parallelized
        do k = 1, n
          a(k) = b(k) + 2.0
        end do
        do j = 1, n
          c(i,j) = a(j) + 2.3
        end do
      end do
      end
```

`f77` automatically eliminates the apparent dependencies here → and here → by making and using its own private versions of array `a()`.

Above, inner loops are *not* parallelized—we do not do both inner and outer.

Exceptions for Automatic Parallelizing

For *automatic* parallelization, the compiler does not parallelize a loop if any of the following occur:

- The step size of the `DO` loop is a variable. (A warning message is issued.)
- The `DO` loop is nested inside another `DO` loop that is parallelized.
- Flow control allows jumping out of the `DO` loop.
- There is a user-level subprogram invoked inside the loop.
- There is an I/O statement in the loop.
- Calculations within the loop change an *aliased scalar* variable.

Concerning Nested Loops

Traditionally, both hand and automatic transformations concentrated on the innermost loop—since performance improvements are multiplied by the number of times the outer loops are executed, that is:

```
      do i                    ! 10 seconds   10k iterations
           do j               ! 10 seconds   10k iterations
               do k           ! 10 seconds   10k iterations
               end do
           end do
      end do
```

On a *single* processing system, improving the "k" loop by 3 seconds would result in the performance being increased considerably more than the "i" loop.

However, on a *parallel* processing system with a relatively small number of processors, it can be most effective to *parallelize* the *outermost* loop. Parallel processing typically involves relatively large loop overheads, so by *parallelizing* the *outermost*, loop we minimize the overhead, and we maximize the work done for each processor.

In general, if there are enough processors, one might want to allocate them *from the top down*. There are many allocation heuristics, some much more complicated than this. The best heuristic requires information about the number of processors, costs of synchronizing parallel threads, and the specific program behavior.

Examples

The following examples illustrate the *definition* of what gets done with *automatic* parallelization, plus the exceptions.

Example: Using `-autopar`, a *call* inside a loop.

```
      ...
      do 40 kb = 1, n              ! ← Not parallelized
           k = n + 1 - kb
           b(k) = b(k)/a(k,k)
           t = -b(k)
           call daxpy(k-1,t,a(1,k),1,b(1),1)
40    continue
      ...
```

Example: Using `-autopar`, a *constant* step size loop.

```
      parameter (del = 2)
      ...
      do k = 3, 1000, del          ! ← Parallelized
           x(k) = x(k) * z(k,k)
      end do
      ...
```

Example: Using `-autopar`, a *variable* step size loop.

```
      integer del / 2 /
      ...
      do k = 3, 1000, del          ! ← Not parallelized
           x(k) = x(k) * z(k,k)
      end do
      ...
```

Example: Using `-autopar`, *nested* loops.

```
      do 900 i = 1, 1000           ! ← Parallelized (outer loop)
           do 200 j = 1, 1000       ! ← Not parallelized (inner loop)
           ...
200        continue
900   continue
```

Example: Using `-autopar`, a *jump out of loop*.

```
      do i = 1, 1000          ! ← Not parallelized
      ...
           if (a(i) .gt. min_threshold ) go to 20
      ...
      end do
20    continue
      ...
```

Example: Using `-autopar`, a loop that conditionally changes a *scalar* variable referenced after a loop.

```
      ...
      do i = 1, 1000          ! ← Not parallelized
           ...
           if ( whatever ) s = v(i)
      end do
      t(k) = s
      ...
```

D.4 Reduction for Automatic Parallelizing

A construct that collapses an array to a scalar is called a *reduction*. Typical reductions are *summing* the elements of a vector, Σv_i, or *multiplying* the elements of a vector, $\prod v_i$. A reduction violates the criterion that calculations

within a loop not change a *scalar* variable in a cumulative way across iterations.
Example: The scalar `s` is changed cumulatively with each iteration.

```
      s = 0.0
      do i = 1, 1000
           s = s + v(i)
      end do
      t(k) = s
```

However, for some constructs, if the reduction is the only thing preventing parallelization, then it is possible to parallelize the construct anyway. Some reductions occur so frequently that it is worthwhile for the compiler to be able to recognize them as special cases and parallelize the constructs.

What You Do

For reduction, use the `-autopar -reduction` option combination.

What the Compiler Does

For reduction, the compiler parallelizes loops that meet the following criteria.

- The programming construct satisfies all the "*Automatic Parallelizing Rules*" except that there is a reduction,
- The reduction is one of the recognized reductions (note list below).

Example: Automatic with reduction, the sum of elements.

```
      s = 0.0
      do i = 1, 1000          ! ← Parallelized
           s = s + v(i)
      end do
      t(k) = s
      ...
demo$ f77 -autopar -reduction
```

Recognized Reductions

Table D-1 Reductions Recognized by the Compiler

Mathematical Entity	**Key FORTRAN Statements**
Sum of the elements	`s = s + v(i)`
Product of the elements	`s = s * v(i)`
Dot product of two vectors	`s = s + v(i) * u(i)`
Minimum of the elements	`s = amin1( s, v(i))`
Maximum of the elements	`s = amax1( s, v(i))`

`OR` of the elements	`do i = 1, n`
	`  b = b .or. v(i)`
	`end do`

`AND` of nonpositive elements	`b = .true.`
	`do i = 1, n`
	`  if (v(i) .le. 0) b=b .and. v(i)`
	`end do`

Count nonzero elements	`k = 0`
	`do i = 1, n`
	`  if ( v(i) .ne. 0 ) k = k + 1`
	`end do`

D

Roundoff and Overflow/Underflow for Reductions

Results from reductions with sums or products of floating-point numbers can be indeterminate. This is so for the following reasons.

- In distributing the calculations over the several processors, the compiler and the runtime environment determine the *order* of the calculations.
- The *order* of calculation affects the sum or product of floating-point numbers, that is, computer floating-point addition and multiplication are not associative. They are not associative because you can get (or not get) roundoff, overflow, or underflow, depending on how you associate the operands. That is, `(X*Y)*Z` and `X*(Y*Z)` may not have the same roundoff, overflow, or underflow.

In some situations the error is acceptable and in others it is not, so use reduction with discretion, depending on your application.

Example: Overflow, underflow, *with*/*without* reduction. PARALLEL is set to 2.

```
demo$ cat t3.f
      real A(10002), result, MAXFLOAT
      MAXFLOAT = r_max_normal()
      do 10 i = 1 , 10000, 2
           A(i) = MAXFLOAT
           A(i+1) = -MAXFLOAT
10    continue

      A(5001)=-MAXFLOAT
      A(5002)=MAXFLOAT

      do 20 i = 1 ,10002 ! Add up the array
           RESULT = RESULT + A(i)
20    continue
      write(6,*) RESULT
      end
demo$ f77 -silent -autopar t3.f
demo$ a.out
   0.


demo$ f77 -silent -autopar -reduction t3.f
demo$ a.out
  Inf
demo$
```

Without reduction. →

0. is correct.

With reduction. →

Infinity is ***not*** correct.

Example: Roundoff. Get the sum of 100,000 random nos. between -1 and +1.

```
demo$ cat t4.f
      parameter ( n = 100000 )
      double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
      s = d_lcrans ( v, n, lb, ub ) ! Get n random nos. between -1 and +1
      s = 0.0
      do i = 1, n
           s = s + v(i)
      end do
      write(*, '(" s = ", e21.15)') s
      end
demo$ f77 -autopar -reduction t4.f
```

Results vary with the number of processors, as shown in the following table.

Sum of 100,000 random nos. between -1 and +1.

Number of Processors	Output
1	s = 0.568582080884714E+02
2	s = 0.568582080884722E+02
3	s = 0.568582080884721E+02
4	s = 0.568582080884724E+02

In this situation the roundoff error is acceptable (on the order of 10^{-14} for data that is random to begin with). Read *What Every Computer Scientist Should Know About Floating-point Arithmetic*" by David Goldberg, in the on-line READMEs directory.

D

iMPact: Explicit Parallelization E

The appendix is organized into the following sections.

What You Do	page 325
What the Compiler Does	page 326
Parallel Pragma	page 327
doall Loops	page 327
Exceptions for Explicit Parallelizing	page 329
Risk with Explicit: Nondeterministic Results	page 333
Scope of Variables for Explicit Parallelizing	page 334
Signals	page 335

This appendix shows a way to parallelize programs for multiple processors. It is called *explicit parallelizing*. It may be faster, with risk of incorrect results.

E.1 What You Do

To parallelize *explicit loops*, do the following.

- Analyze loops to detect those with no order-dependence. This requires far more analysis and sophistication than using automatic parallelization.
- Insert a special directive *just before* each loop that you want parallelized.
- Use the `-explicitpar` option on the `f77` command line.
- Check results very carefully.

The special directive "`c$par doall`"is described later, but first it is illustrated in the following example.

Example: Parallelize *explicitly*, the "`DO I=1, N`" loop.

See Appendix C, "iMPact: Multiple Processors for required background.

```
c$par doall
      do i = 1, n          ! This loop gets parallelized.
           a(i) = b(i) * c(i)
      end do
      do k = 1, m          ! This loop does not get parallelized.
           x(k) = y(k) * z(k)
      end do
demo$ f77 -explicitpar t1.f
```

No Primitives

If you do your own multi-threaded coding using the `libthread` primitives, do *not* use `-explicitpar`. Either do it all yourself or let the compiler do it. Conflicts and unexpected results may happen if you and the compiler are both trying to manage threads with the same primitives.

See *–mt* in Chapter 3, "Using the Compiler."

E.2 What the Compiler Does

For *explicit* parallelization, the compiler parallelizes those loops that *you* have specified. This is similar to the transformations of a vectorizing compiler.

The compiler applies appropriate dependence-based restructuring transformations. It then distributes the work evenly over the available processors. Each processor executes a different chunk of iterations.

Example: 4 processors, 1000 iterations; the following occur simultaneously.

Processor 1 executing iterations	1	through	250
Processor 2 executing iterations	251	through	500
Processor 3 executing iterations	501	through	750
Processor 4 executing iterations	751	through	1000

E.3 Parallel Pragma

*Explicit*ly parallelizing loops requires using a *parallel pragma* and command-line *options*. A *parallel pragma* is a special comment that directs the compiler to do some parallelizing. Pragmas are also called *compiler directives*. Currently there is only one parallel pragma, the `doall` pragma.

- `c$par doall`

P*arallel* pragmas have the following syntax:

- First character is in column one (only parallel pragmas require this)
- First character can be any one of `c`, `C`, `d`, `D`, `*`, or `!`
- Next 4 characters must be `$par`, no blanks, any uppercase and lowercase

For `doall`, the compiler parallelizes the *next loop* it finds after the pragma, if possible.

A loop with an explicit pragma is excluded from any automatic reductions.

E.4 `doall` *Loops*

To use explicit parallelization safely, you must understand the rules for explicit parallelizing. Explicit parallelization of a `doall` loop requires far more analysis and sophistication than *automatic* parallelization. There is far more risk of indeterminate results. This is not only roundoff, but inter-iteration interference.

Definition

For explicit parallelization the `doall` loop is defined as follows:

- The construct is a `DO` loop (uses the `DO` statement, but not `DO WHILE`).
- The values of *array* variables for each iteration of the loop `do` not depend on the values of *array* variables for any other iteration of the loop.
- Calculations within the loop do not change any *scalar* variable that is referenced *after* the loop terminates. Such scalar variables are not guaranteed to have a defined value after the loop terminates, since the compiler does not ensure a proper storeback for them.
- For each iteration, any *subprogram* invoked inside the loop does not reference or change values of *array* variables for any other iteration.

Explicitly Parallelizing a `doall` *Loop*

To explicitly parallelize a `doall` loop, do the following.

- Use the `-explicitpar` option on the `f77` command line.
- Insert a `doall` parallel pragma immediately before the *specific* loop.

Example: Explicit, `doall` loop.

```
demo$ cat t4.f
     ...
c$par doall
     do i = 1, n                    ! ← Parallelized
          a(i) = b(i) * c(i)
     end do

     do k = 1, m                    ! ← Not parallelized
          x(k) = x(k) * z(k,k)
     end do
     ...
demo$ f77 -explicitpar t4.f
```

Example: Explicit, `doall`, some calls can make dependencies.

```
demo$ cat t5.f
     ...
c$par doall
     do 40 kb = 1, n                ! ← Parallelized
          k = n + 1 - kb
          b(k) = b(k)/a(k,k)
          t = -b(k)
          call daxpy(k-1,t,a(1,k),1,b(1),1)
40   continue
     ...
demo$ f77 -explicitpar t5.f
```

The code is taken from `linpack`. The subroutine `daxpy` was analyzed by some software engineer for iteration dependencies and found to *not* have any. It is a nontrivial analysis.

This example is an instance where explicit parallelization is useful over automatic parallelization.

CALL *in a Loop*

It is sometimes difficult to determine if there are any inter-iteration dependencies. A subprogram invoked from within the loop requires advanced dependency analysis. Since such a case works only under explicit parallelization, it is *you* who must do the advanced dependency analysis, not the compiler.

The following rule sometimes helps with subprogram calls in a loop:

> Within a subprogram, if all local variables are *automatic*, rather than *static*, then the subprogram does not have iteration dependencies.

Note that the above rule is sufficient, but it is by no means necessary. For instance, the `daxpy()` routine in the previous example does not satisfy this rule, and it does not have iteration dependencies, although that is not obvious.

You can make all *local* variables of a subprogram automatic as follows:

- List them in an `automatic` statement. However, then you cannot initialize them in a `data` statement.
- Compile the subprogram with the `-stackvar` option.

E.5 Exceptions for Explicit Parallelizing

In general, the compiler parallelizes a loop if you explicitly direct it to, but there are exceptions—some loops the compiler just cannot parallelize.

The following are the primary detectable exceptions that may prevent explicitly parallelizing a `DO` loop. Examples of each are provided below.

- The `DO` loop is nested inside another `DO` loop that is parallelized.

  This exception holds for indirect nesting, too. If you explicitly parallelize a loop, and it includes a call to a subroutine, then even if you parallelize loops in that subroutine, still, at runtime, those loops are not run in parallel.
- The step size parameter is a variable.
- A flow control statement allows jumping out of the `DO` loop.
- The index variable of the loop is subject to side effects, such as being equivalenced.

Warning Messages by `-vpara`

If you compile with `-vpara`, you may get a warning message if f77 detects a problem with explicitly parallelizing a loop. f77 may still parallelize the loop.

Table E-1 Exceptions for Explicit Parallelizing

Exceptions that May Prevent Explicitly Parallelizing Loops	**Parallelized**	**Message**
Loop is nested inside another loop that is parallelized	No	No
Loop is in a subroutine, and a call to the subroutine is in a parallelized loop	No	No
Step size parameter is a variable	No	No
Jumping out of loop is allowed by a flow control statement	No	Yes
Index variable of loop is subject to side effects	Yes	No
Some variable in the loop keeps a loop-carried dependency	Yes	Yes
I/O statement in the loop—*usually unwie, because the order of output is random*	Yes	No

Example: Nested loops, not parallelized, no warning.

```
     ...
c$par doall
     do 900 i = 1, 1000            ! ← Parallelized (outer loop)
          do 200 j = 1, 1000        ! ← Not parallelized--no warning
          ...
200       continue
900  continue
     ...
demo$ f77 -explicitpar -vpara t6.f
```

Example: Loop in subroutine; a call to it is in a parallelized loop, not parallelized, no warning.

```
c$par doall
     do 100 i = 1, 200
          ...
          call calc (a, x)
          ...
100  continue
     ...
demo$ f77 -explicitpar -vpara t7.f
```

↑ At runtime this loop may run in parallel.

```
     subroutine calc ( b, y )
     ...
c$par doall
     do 1 m = 1, 1000
          ...
1    continue
     return
     end
```

↑ At runtime, loops do *not* run in parallel.

Example: Variable step size, not parallelized, no warning.

```
     integer del / 2 /
     ...
c$par doall
     do k = 3, 1000, del          ! ← Not parallelized --no warning
          x(k) = x(k) * z(k,k)
     end do
     ...
demo$ f77 -explicitpar -vpara t8.f
```

Example: Jumping out of loop, not parallelized, warning.

```
c$par doall
     do i = 1, 1000          ! ← Not parallelized--warning
     ...
          if (a(i) .gt. min_threshold ) go to 20
     ...
     end do
20   continue
     ...
demo$ f77 -explicitpar -vpara t9.f
```

Example: Index variable subject to side effects, parallelized, no warning. .

```
     equivalence ( a(1), y )       ! ← Source of possible side effects
     ...
c$par doall
     do i = 1, 2000              ! ← Parallelized--no warning, but unsafe
          y = i
          a(i) = y
     end do
     ...
demo$ f77 -explicitpar -vpara t11.f
```

Example: Variable in loop has loop-carried dependency, parallelized, warning

```
c$par doall
     do 100 i = 1, 200          ! ← Parallelized--warning
          y = y * i             ! ← y has a loop-carried dependency
          a(i) = y
100  continue
     ...
demo$ f77 -explicitpar -vpara t12.f
```

I/O with Explicit Parallelization

The library `libF77_mt` is mt-safe, but mostly not mt-hot. It is OK to do I/O in a loop that will be executed in parallel, provided:

- You do not care that the output from different threads will be interleaved, so program output is non-deterministic
- You take responsibility for the safety of executing the loop in parallel, because you must use an explicit directive and `-explicitpar` (or `-parallel`) option.
- You realize that a loop which contains I/O will never be automatically parallelized. So don't do I/O in loops you want to be considered for automatic parallelization.

Example: I/O statement in loop, parallelized, no warning (*usually unwise*).

```
c$par doall
      do i = 1, 10          ! ←  Parallelized--no warning ---unwise
           k = i
           call show ( k )
      end do
      subroutine show( j )
      write(6,1) j
1     format('Line number ', i3, '.')
      end
demo$ f77 -silent -explicitpar -vpara t13.f
demo$ setenv PARALLEL 2
demo$ a.out     {This displays the numbers 1 through 10, but in a different order each time.}
```

Example: Recursive I/O hangs.

It is not OK to do recursive (nested) I/O when you have compiled with `-mt`.

This program deadlocks in `libF77_mt`, and hangs.

Hit Control-C key to regain keyboard control

```
      do i = 1, 10          ! ←  Parallelized--no warning ---unsafe
           k = i
           print *, list( k ) ! list is a function that does I/O
      end do
      end
      function list( j )
      write(6,"('Line number ', i3, '.')") j
      list = j
      end
demo$ f77 -mt t14.f
demo$ setenv PARALLEL 2
demo$ a.out
```

E.6 Risk with Explicit: Nondeterministic Results

A set of operations can be safely executed in parallel only if the computational result does not depend on the order of execution. For *explicit* parallelizing, *you* (rather than the compiler) specify which constructs to parallelize, and then the compiler parallelizes the specified constructs. You do your own *dependency analysis.*

If you force parallelization where dependencies are real, then the results depend on the order of execution; they are *nondeterministic*; you can get incorrect results.

Testing is not Enough

An entire test suite can produce correct results over and over again, and then produce incorrect results. What happens is that the number of processors (or the system load, or some other parameter) changed. So you must test with different numbers of processors, different system loads, and so forth. But this means you cannot be exhaustive in your test cases.

The problem is *not* roundoff but interference between iterations. An example of this is one iteration referencing an element of an array that is calculated in another iteration, but the reference happens before the calculation.

One approach is systematic analysis of every explicitly parallelized loop. To be sure of correct results, you must be certain there are no dependencies.

Example: Loop with dependency: parallelize explicitly, *nondeterministic* result.

```
      real a(1001), s / 0.0 /
      do i = 1, 1001     ! Initialize array a.
           a(i) = i
      end do
c$par doall
      do i = 1, 1000     ! This loop has dependencies.
           a(i) = a(i+1)
      end do
      do i = 1, 1000     ! Get the sum of all a(i).
           s = s + a(i)
      end do
      print *, s         ! Print the sum.
      end
demo$ f77 -explicitpar t1.f
```

How Indeterminacy Arises

In a simpler example, 4 processors, 8 iterations, same kind of initialization:

- The first 2 iterations run on processor 1
- The next 2 iterations run on processor 2
- …

All processors run simultaneously, and *usually* finish at about the same time. But the compiler provides no synchronization for arrays, and for many reasons, one processor *can* finish before others; you cannot know the finishing order in advance.

Processor 1	Processor 2	Processor 3	Processor 4
`a(1) = a(2)`	`a(3) = a(4)`	`a(5) = a(6)`	`a(7) = a(8)`
`a(2) = a(3)`	`a(4) = a(5)`	`a(6) = a(7)`	`a(8) = a(9)`

When processor 1 does `a(2) = a(3)`:

- If processor 2 has done `a(3) = a(4)`, then `a(2)` gets 4
- If processor 2 has *not* yet done `a(3) = a(4)`, then `a(2)` gets 3

Therefore the values in `a(2)` depend on which processor finishes first. After completion of the parallelized loop, the values in array `a` depend on which processor finishes first. And which finishes second, … So the sum depends on events you cannot determine. The major variables in the runtime environment that cause this kind of trouble are the number of processors in the system, the system load, interrupts, and so forth. However, you usually cannot know them *all*, much less control them all.

E.7 Scope of Variables for Explicit Parallelizing

It sometimes helps to know how the compiler treats some kinds of variables. It distinguishes between *scalar* and *array*, and between *local* and *shared* variables.

Definitions

- For the purposes of this analysis, a *local* variable is one that is *private to a single iteration* of a loop. The value assigned to a local variable in one iteration is not propagated to any other iteration of the loop.

- A *shared* variable is one that is shared with all other iterations. The value assigned to a shared variable in an iteration is seen by other iterations of the loop. If an explicitly parallelized loop contains shared references, then you must ensure that sharing does not cause correctness problems. The compiler does no synchronization on updates or accesses to shared variables.

For loops with explicit directives, the compiler uses certain rules to determine whether a variable is shared or local. (There is no mechanism to specify the attributes of variables referenced inside a loop.)

Rules

1. All *scalar* variables are treated as *local* variables. This means a processor local copy of the scalar variable is made in each processor and that local copy is used within that process.
2. All *array* references are treated as *shared* variable references. This means any write of an array element by one processor is visible to all processors. No synchronization is performed on accesses to shared variables.
3. If inter-iteration dependencies exist in a loop, then the execution may result in erroneous results. It is your responsibility to ensure that these cases do not arise. The compiler may sometimes be able to detect such a situation at compile-time, and issue a warning. However, it will not disable parallelization of such loops.

E.8 Signals

In general, if the loop you are parallelizing does any signal handling, then there is a risk of unpredictable behavior, including a system hang, getting hosed, and other generic bad juju.

In particular, if

- The I/O statement raises an exception
- The signal handler you provide does I/O

then your system can lock up. This causes problems even on single-processor machines.

Two common ways of doing signal handling without being explicitly aware of it are the following.

- Input/Output statements (`WRITE`, `PRINT`, and so forth) that raise exceptions
- Requesting Exception Handling

Example: Output that can raise exceptions.

```
      real x / 1.0 /, y / 0.0 /
      print *, x/y
      end
```

Input/Output statements do locking, and if an exception is raised then there may be an attempt to lock an already locked item, resulting in a deadlock.

One (possibly overly cautious) approach: If you are parallelizing, do not have I/O in that loop, and do not request exception handling.

Example: Using a signal handler which breaks the rules.

```
      character  string*5, out*20
      double precision value
      external exception_handler

      print *, ' '
      print *, 'output'
      i = ieee_handler('set', 'all', exception_handler)
      read(5, '(E5)') value
      string = '1e310'
      read(string, '(E5)') value
      print *, 'Input string ', string, ' becomes: ', value
      print *, 'Value of 1e300 * 1e10 is:', 1e300 * 1e10
      i = ieee_flags('clear', 'exception', 'all', out)
      end

      integer function exception_handler(sig, code, sigcontext)
      integer sig, code, sigcontext(5)
      print *, '*** IEEE exception raised!'
      return
      end
```

Index

Symbols

#include path, 41
%VAL(), pass by value, 241
.F suffix, 23, 102
.fln files
    directory, -Xlist, 149
    -Xlist, 143
/usr/ccs/lib, error to specify it, 44
/usr/lib, error to specify it, 43
/usr/lib, never use -L*dir*, 118
_
    do not append _ to external names, 67, 241

Numerics

132-column lines, -e, 35
2-byte integers, 40
-386, 29
-486, 29
4-byte integers, 40
80-column lines, -e, 35
8-bit
    characters, 63
    clean, 63

A

-a, 30
a.out file, 22
abort, 37
    on exception, 37
abrupt underflow, 38, 194
access
    named files, 85
    on multifile tapes, 97
    unnamed files, 88
accrued exceptions, do not warn, 180
actions / options sorted by action, 26
addenda for manuals, read me file, xxv
agreement across routines, -Xlist, 139
alarm(), do not call from MP, 10
alias, 100
    many options, short commands, 69
-align, 70
align
    block, -align workaround, 70
    data types, 239
    double word, -dalign, 33
    errors across routines, -Xlist, 139
    page boundary, -align, 30, 70
    structures as in prior releases, 48
    structures as in VMS, 59
-align, 30

analysis files, `.fln`, `-Xlist`, 143
analyzer compile option, `-xF`, 58
ANSI, 3
  conformance check, `-Xlist`, 141
`-ansi` extensions, 30
AnswerBook, xxiii
  documents in, xxiii
`ar`, 122
  create static library, 123
arithmetic
  nonstandard, 37, 194
  standard, 194
array
  bounds, 32
  bounds, exceeding, 159
  C FORTRAN differences, 242
  `dbx`, 164, 165
  slices in `dbx`, 165
`asa` FORTRAN print, 15
attributes XView, 294
audience, xxii
autoload
  `dbx`, disable, 61
  definition, 61
automatic
  parallelization
    definition, 316
    exceptions, 317
    overview, 304
    usage, 313
    what the compiler does, 314
  variables, 55
`-autopar`, parallelize automatically, 31
autovectorizing compiler,
    comparison, 307

B

backslash, 59
`BaseDir`, base directory, 120
basic block, profile by, `-a`, 30
`-Bdynamic`, 32
benchmark case history, 230
best
  floating point `-native`, 45
  performance, 49
binding
  dynamic, 32, 34
  static, 32
bindings
  POSIX, 135
  Xlib, 135
  XView, 135
boldface font conventions, xxvi
bounds of arrays, 32, 153
  checking, 159
box
  clear, xxvi
  indicates nonstandard, xxvi
browser, 55
BS 6832, 3
`-bsdmalloc`, 31
`-Bstatic`, 32
bus error
  locating, 162
  some causes, 162

C

C, 257, 273
  calls FORTRAN, 268
  directive, 67, 241
  is called by FORTRAN, 245
  pragma, 67
  preprocessor, 38, 102
`-C`, 153, 160
  check subscripts, 32
C FORTRAN
  function compared to subroutine, 238
  key aspects of calls, 237
  labeled common, 264, 279
  sharing I/O, 265, 280
`-c`, compile only, 32
cache, fast, Solaris 1.x, 116

call
C from FORTRAN, 245
FORTRAN from C, 268
graphs, -Xlistc, 148
CALL in a loop, parallelize, 55
case preserving, 57, 154, 240
catalog, 14
Catalyst, 14
catch FPE, 161, 195, 196
-cg89, 32
-cg92, 32
change a constant, 9
check
strictness, -Xlist, 150
subscripts, -C, 32
clear box, xxvi
code generator option, -cgyr, 32
command
ar, create static library, 123
asa, 15
compiler, 22
dd, conversion utility, 96
f77, 22
fsplit, 71
ranlib, randomize library, 123
comments
debug, VMS, 60, 155
to Sun, xxv, 40
common block
maps, -Xlist, 150
page-alignment, 30, 70
compatibility
FORTRAN 2.0/2.0.1 source with FORTRAN 3.0/3.0.1, 13
none for FORTRAN 1.4 binaries with FORTRAN 2.X, 13
compile
check across routines, -Xlist, 142
fails, message, 22
link for a dynamic shared library, 39
link sequence, 22
link, consistent, 116
make assembler source files only, 56
compile (*continued*)
make source listing with diagnostics, 168
off, do cpp only, -F, 38
only, -c, 32
passes, times for, 57
search path directory, 52
compile action
2-byte integers, -i2, 40
4-byte integers, -i4, 40
align
common blocks, -align, 30, 70
on 8-byte boundaries, -f, 36
structures the old way, -oldstruct, 48
analyze threads, -Ztha, 63
ANSI, show non-ANSI extensions, -ansi, 30
assembly-language output files, keep, -S, 56
autoload, disable for dbx, -xs, 61
automatic parallelization, -autopar, 31
blank in column one, none, list-directed output, -oldldo, 48
C preprocessor, 38
check
across routines, -Xlist, 62
subscripts, -C, 32
compile only, -c, 32
debug
-g, 39
statement, VMS, -xld, 60
define name for cpp, -D*name*, 34
dependency-based scalar optimization in loops, -depend, 33
disable autoload for dbx, 61
DO loops for one trip min, -onetrip, 48
double, interpret real as double precision, -r8, 53

compile action (*continued*)
dynamic binding
-Bdynamic, 32
-dy, 34
executable file, name the, -o outfil, 48
explicit parallelization, -explicitpar, 35
extend lines to 132 columns, -e, 35
extend the language, VMS, 59
fast
execution, -fast, 36
global checking, -Xlistf, 149
malloc, 31
sourcebrowser, -sbfast, 55
feedback to Sun, -help, 40
floating point
best, -native, 45
nonstandard, -fnonstd, 37
force floating-point precision of expression, 38
function-level reordering, -xF, 58
generate code for
80386, -386, 29
80486, -486, 29
generic SPARC, -cg89, 32
Pentium, -pentium, 50
SPARC, V8 -cg92, 32
generate double load / store instructions, -dalign, 33
global program checking, -Xlist, 62
inline templates
off, -nolibmil, 46
select best, -libmil, 42
inline the specified user routines, 40
library
add to search path for, -Ldir, 43
build shared library, -G, 39
name a shared dynamic, -hname, 39
license
do not queue request, -noqueue, 46
information, -xlicinfo, 60
link with library x, -lx, 43
list of options, -help, 40

compile action (*continued*)
list-directed output, old, -oldldo, 48
loops, show which loops are parallelized, 42
math speed, use selected math routines optimized for performance, 60
misaligned data, -misalign, 44
MT, use multi-thread safe libraries, 44
no automatic libraries, 46
no automatic parallelization, 45
no -depend, 45
no explicit parallelization, 45
no forcing of expression precision, 46
no reduction, 47
no run path, 47
optimize object code, -On, 49
parallelize, -parallel, 50
pass option to other program, -Qoption, 52
paths, store into object file, 54
print
name of each pass as compiler executes, -v, 57
version id of each pass as compiler executes, -V, 58
produce
position-independent code, -PIC, 51
position-independent code, -pic, 51
source code of specified type, -Qproduce, 52
profile by
loop, MP, -Zlp, 62
procedure, -p, 50
procedure, -pg, 51
statement, -a, 30
quiet compile, 55
reduction, analyze loops for reduction, 53
report execution times for compilation passes, -time, 57

compile action (*continued*)
reset -fast so that it does not use `-xlibmopt`, 60
resize static compiler tables, `-N`, 47
set
`#include` path, `-I`path, 41
directory for temporary files, `-temp`*dir*, 56
level of checking strictness, `-Xlistv`*n*, 150
nesting level of
control structures, 47
data structures, 47
number of
continuation lines, 47
equivalenced variables, 48
external names, 48
identifiers, 47
statement numbers, 48
search path compilation directory, `-Qpath`, 52
short integers, `-i2`, 40
show commands, `-dryrun`, 34
simple floating-point mode, 38
smaller executable file, 45
source browser, prepare for, `-sb`, 55
stack the local variables, 55
standard integers, `-i4`, 40
static binding
`-Bstatic`, 32
`-dn`, 34
strip executable file of symbol table, 55
undeclared, make default type undeclared, `-u`, 57
unroll loops, 57
uppercase in variable names, `-U`, 57
verbose
parallelization warnings, `-vpara`, 57
`-v`, 57
VMS features, `-xl`, 59
warnings, suppress all `f77` warning messages, `-w`, 58
compile option differences for
Solaris 2.x, 1.x, x86, 7
compiler
command, 22
XView, 290
error messages in local language, 64
frequently used options, 25
passes, 57
recognizes files by types, 24
tables, 47
complete path name, 80
consistent
across routines, `-Xlist`, 139
arguments, commons, parameters, etc., 62
compile and link, 23, 116
compile options, 23, 53, 62
constant, trying to change a constant, 9
continuation lines, number of, 47
control structure level, 47
conventions in text, 4
Courier font, xxvi
`cpp`, the C preprocessor, 23, 102
create
library, 122
library, dynamic, 128
SCCS files, 107
cross reference table, `-Xlist`, 62, 151
current working directory, 79

D

`d`
comment line debug statements, VMS, 155
in column one, 60
`-D` option, define name for `cpp`, 103
`-dalign`, 33
data
inspection, `dbx`, 167
structure levels, 47
types XView, 296
`date`, 135

dbx, 155
  arrays, 164
  catch FPE, 160, 161, 196
  commands, 166
  current procedure and file, 166
  debug, 15
  f77 -g, 39
  -g, 157
  initializes faster, 61
  language command, 10
  locate exception, 196
    by line number, 160, 161
  next, 159
  print, 158
  quit, 157
  run, 158
  set breakpoint, 157
dd conversion utility, 96
debug, 139, 195
  arguments, agree in number and type, 139
  array, 164
    print row or column, 165
    slices, 165
  block data, 13
  case-sensitive compiles, -U, 57
  checking across routines for global consistency, 139
  column print, array, 165
  comments, VMS, 60
  common blocks, agree in size and type, 139
  compiler options, 153
  dbx, 15
  debugger, 15
  disable autoload for dbx, 61
  IEEE exceptions, 195
  locating exception, 196
    by line number, 161
  option, 39
  parallelized code, 309
  parameters, agree globally, 139
  row print, array, 165
  sbrowser, 15
  slices of arrays, 165
debug (*continued*)
  tips for parallelized code, 309
  uppercase, 166
  with optimized code, 12
  with other languages, 10
debugger, main features, 167
debugging
  aids, linker, 115
  parallelized programs, 309
declared but unused, checking, -Xlist, 141
deep, vasty, 235
default
  size
    complex, 53
    integers, 40
    logicals, 40
    reals, 53
  type undeclared, 57
define name for cpp, -Dname, 34
delete .fln files, 143
-depend, scalar optimization, 33
dependency
  analysis, 315
  analysis -depend, 33
  with explicit parallelization, 333
depth for
  control structures, 47
  data structures, 47
diagnostics, source, 168
diamond indicates nonstandard, xxvi
direct I/O, 92
directive, 67
directory, 79
  .fln files, 143
  compilation search path, 52
  current working, 79
  object library search, 43
  temporary files, 56
display to terminal, -Xlist, 142
division by zero IEEE, 175
dmesg, actual real memory, 73
-dn, 34

DO loops executed once, -onetrip, 48
doall
  loop, 327
  pragma, 68, 327
documents on-line, xxiii
double quote, 59
double-word align, 33
-dryrun, 34
dtime in MP, 12
-dy, 34
dynamic
  binding, 34
  library, 125
    advantages, disadvantages, 126
    build, -G, 39
    create
      Solaris 1.x, 130
      Solaris 2.x, 128
    initialized data, Solaris 1.x, 131
    name a dynamic library, 39
    path in executables, 54
  show if a.out is dynamically linked, 134

E

-e, extended source lines, 35
ed, 14
email
  alias, Sun Programmers SIG, 357
  send feedback comments to Sun, xxv
environment
  getenv, 86
  variable, shorten command line, 69
EOS package, 14
equivalence block maps, -Xlist, 150
equivalenced variables, number of, 48
era, 14
errata and addenda for manuals, read me file, xxv
error
  messages, 283
    in the local language, 64
    with source listing, error, 168
  standard error, 83, 89
    accrued exceptions, 194
  utility, 168
errors only, -XlistE, 148
establish a signal handler, 186
event management, dbx, 167
ex, 14
exceptions
  accrued, 181
  detect
    all 5 IEEE, 182
    all 5, ieee_handler, 189, 190
    by signal handler, 186, 196
  explicit parallelization, 329
  handlers, 176, 184
  ieee_handler, 184
  location in dbx, 196
    by line number, 161
  unrequited, 194
executable file
  built-in path to dynamic libraries, 54
  dynamically linked?, 134
  generating it, 23
  names in, nm command, 124
  naming it, 48
  strip symbol table from, 55
execution time
  compilation passes, 57
  optimization, 49
explicit
  parallelization, 325
    exceptions, 329
    overview, 304
    risk, 333
  typing, 57
-explicitpar, parallelize explicitly, 35
export initialized data from dynamic library, Solaris 1.x, 132

extended
language -xl, 59
lines -e, 35
syntax check, -Xlist, 141
extensions
non-ANSI, 30
VMS features with -xl, 59
external
C functions, 67, 241
names, 240
names, number of, 48

F

-F, 38
F file suffix, 23
-f, align on 8-byte boundaries, 36
f_exit(), 280, 281
f_init(), 280, 281
f77, 22
-fast
fast execution, 36
no libm.il, 46
fast cache, Solaris 1.x, 116
faster
linking and initializing, 61
malloc, 31
output, global checking, -Xlistf, 149
features
debugger, 167
new or changed, 5
VMS, with -xl, 59
feedback file for email to Sun, xxv
feedback to Sun, -help, 40
FFLAGS shorten command line, 69
file
.fln
directory, -Xlist, 149
-Xlist, 143
a.out, 22
directory, 79
executable, 22
information files, xxv
file (*continued*)
internal, 93
object, 22
permissions C FORTRAN, 244
pipe, 83
preattached, 90
redirection, 82
size too big, 71
split by fsplit, 15
standard
error, 89
input, 89
output, 89
standard error, 89
system, 77
file command, 134
file names, 86
passing to programs, 88
recognized by the compiler, 24
files and optimization, 234
FIPS 69-1, 3
fix and continue, dbx, 167
-flags, 37
floating-point
Goldberg paper, xxiii
hardware, 69
nonstandard initialization, 37
option, -native, 45
-fnonstd, 37
font
boldface, xxvi
conventions, xxvi
Courier, xxvi
italic, xxvi
FORTRAN
calls C, 245
is called by C, 268
MP, 303
read me file, bugs, new/changed features, xxv
four-byte integers, 40
FPE catch in dbx, 161, 196
fpversion, show floating-point version, 69

-fsimple, simple floating-point model, 38
fsplit, 71
  FORTRAN file split, 15
-fstore, 38
function
  called within a loop, parallelization, 329
  compared to subroutine, C FORTRAN, 238
  data type of, checking, -Xlist, 141
  external C, 67
  library, 135
  names, 240
  return values
    from C, 257
    to C, 273
  unused, checking, -Xlist, 141
  used as a subroutine, checking, -Xlist, 141
function-level reordering, 58

G

-G, 39
-g, 39
gencat, 65
generic procedures for XView, 293
getc library routine, 97
getcwd, 79
getenv environment, 86
Glendower, 235
global
  optimization, 4, 49
  program checking, 139
Goldberg, floating-point white paper, xxiii
gprof
  -pg, profile by procedure, 51
  profile by procedure, 15
  usage, 218
gradual underflow, 194
graphically monitor variables, dbx, 167
GSA validated, 3
guidelines for number of processors, 308

H

-h name, 39
handlers, exception, 176, 184
handles, XView, 296
hardware
  floating-point fpversion, 69
  floating-point nonstandard initialization, 37
header files for XView, 292
-help, 40
Henry IV, 235
hierarchical file system, 77
Hotspur, 235

I

I/O, 82, 223
-i2, short integers, 40
-i4, 40
idate, 135
identifiers, number of, 47
IEEE, 175, 194, 195
  754, 3
  exceptions, 176
  signal handler, 186
  warning messages off, 180
ieee_flags, 177, 179
ieee_functions, 177
ieee_handler, 177, 184
ieee_values, 177, 183
impatient user's guide, 17
implicit typing, off, 57
INCLUDE, 91
incompatibility FORTRAN 1.4 binaries with FORTRAN 2.X, 13
inconsistency
  arguments, checking, -Xlist, 141
  named common blocks, checking, -Xlist, 141

increase stack size, 56
indeterminacy, how it arises, 334
index check of arrays, 153, 159
inexact exception, 193
information files, xxv
initialize
I/O for FORTRAN from C, 280
nonstandard floating-point hardware, 37
initialize data, dynamic library, Solaris 1.x, 131
inline, 36
code and optimization, 234
templates none, `-nolibmil`, 46
templates, `-libmil`, 42
user-written routines, 40, 59
`-inline`, 40
input
output, initialize for FORTRAN from C, 280
redirection, 82
standard, 89
inserting SCCS ID keywords, 106
integer, size four bytes, 40
interface
for C and FORTRAN, 235
problems, checking for, `-Xlist`, 141
internal files, 93
internationalization, 63
interpret `REAL` as `DOUBLE PRECISION`, 53
invalid, IEEE exception, 175
`IOINIT`, 11, 90
`iostats`, 223
`-I`path, 41
italic font conventions, xxvi

K

`-KPIC`, 42
`-Kpic`, 42

L

labeled common C FORTRAN, 264, 279
labels, unused, `-Xlist`, 141
language
extended `-xl`, 59
local, 64
preprocessor, 23
`language` command, `dbx`, 10
large files, 71
`LC_MESSAGES`, 66
`LD_LIBRARY_PATH`, 117, 118, 119, 121
`LD_RUN_PATH`, 121
`LD_RUN_PATH` and `-R`, not identical, 54
`ldd` command, 134
`-L`dir, 43
level of
checking strictness, `-Xlistv`*n*, 150
control structure, 47
data structures, 47
`libm`, user error making it unavailable, 43
`-libmil`, 42
libraries
advantages, disadvantages, 114
C FORTRAN, 242
in general, 113
math, 135
order on command line, `-l`x, 119
paths in executables, 54
POSIX, 136
profile missing, 224
search order, 117, 120
used by `a.out`, `file` command, 134
VMS, 135
XView, 290
library
build, `-G`, 39
create, dynamic, 128
create, static, 122
initialized data, Solaris 1.x, 131
load, 43
loaded, 115
name a shared library, 39
not found, 118

library (*continued*)
paths in executables, 54
replace module, 125
shared, 125
static, 122
`libV77`, 135
license
information, 60
no queue, 46
licensing, 16
limit, 72
`limit` stack size, 56
line number of
bus error (`SIGBUS`), 162
exception, 161
segmentation fault (`SIGSEGV`), 159
line width, output, `-Xlist`, 151
line-numbered listing, `-Xlist`, 142
lines extended `-e`, 35
link
options, 116
sequence, 22
suppress, 32
linker, 23
links faster, 61
search order, 117
lint-like checking across routines, `-Xlist`, 139
list of options, 40
listing
line numbered with diagnostics, `-Xlist`, 139
with diagnostics, `error`, 168
`-Xlist`, 149
load
library, 43
map, 115
loaded library, 115
loader, 23
loading more slowly, 9
local
language, 64
variables, 55
locating
bus error by line number, 162
exception, 196
by line number, 161
segmentation fault by line number, 160
logical
file names, 59
file names in the `INCLUDE`, 91
size four, 40
unit preattached, 90
long command lines, 69
loop
dependence analysis, `-depend`, 33
jamming, 229
parallelizing a `CALL` in a loop, 55
profiling, 62
restructuring, `-depend`, 33
`-loopinfo`, show which loops are parallelized, 42
`looptool`, loop profiler for MP, 62
lowercase, do not convert to, 57, 240
`-lV77`, 135

M

`-m` linker option for load map, 115
macros
overriding values, 104
with `make`, 103
magnetic tape I/O, 95
main
stack, 56
`make`, 100, 105
making SCCS directory, 106
many options, short commands, 69
maps
common blocks, `-Xlist`, 150
equivalence blocks, `-Xlist`, 150
load, 115
mateo, 163

math
library, 135
library, user error making it unavailable, 43, 117
membership in SunPro SIG, Sun Programmers Special Interest Group, 357
memory
actual real memory, display, 73
limit virtual memory, 72
optimizer out of memory, 71
usage, 217
messages, 283
error, in source listing, 168
local language versions, 64
MIL-STD-1753, 3
miscellaneous tips
alias, many options, short commands, 69
environment variable, many options, short commands, 69
floating-point version, 69
missing
library, 118
profile libraries, 224
-Mmapfile, 58
monitor variables graphically, dbx, 167
MP FORTRAN, 304
-mt, multi-thread safe libraries, 44
multifile tape access, 98
multiplying and reduction, automatic parallelization, 320
multiprocessing standards, 306
multiprocessor FORTRAN, 303
mvbits, 135

N

-N, 47
name
compiler pass, show each, 57
executable file, 48
names in executable, nm command, 124
-native, 36
floating point, 45
native language characters, 63
NBS validated, 3
-Nc, 47
-Nd, 47
nesting
control structures, 47
data structures, 47
parallelized loops, 317, 329
network licensing, 16
NIST validated, 3
-Nl, 47
nm, names in executable, 124
-Nn, 47
no license queue, 46
no such file or directory, cause, 224
-noautopar, 45
-nocx, 45
-nodepend, 45
-noexplicitpar, 45
-nofstore, 46
-nolib, 46
-nolibmil, 46
non-ANSI extensions, 30
nondeterministic results, explicit parallelization, 333
nonstandard
arithmetic, 38, 194
indicated by diamond, xxvi
initialization of floating point, 37
PARAMETER, 59
-noqueue, 46
-noreduction, 47
-norunpath, 47
-Nq, 48
-Ns, 48

number of
bytes of I/O, 223
continuation lines, 47
equivalenced variables, 48
external names, 48
I/O statements, 223
identifiers, 47
processors, 10
for parallelization, 307
reads and writes, 217
statement numbers, 48
swapouts, 217
-Nx, 48

O

-O, 49
with -g, 39, 49
-o, output file, 48
-O1, 49
-O2, 49
-O3, 49
-O4, 49
object library search directories, 43
obscurities, checking for -Xlist, 141
ode to trace, 163
off
autoload for dbx, 61
blank in listed-directed output, 48
converting uppercase letters to lowercase, 57
display of entry names and file names, 55
implicit typing, 57
inline templates for -fast, 46
-lcx, 45
license queue, 46
link system library, 46
linking, 32
underscores, 67, 241
warnings
f77 warnings, 58
IEEE accrued exceptions, 180
-xlibmopt, 60
-oldldo, 48
-oldstruct, 48
-onetrip, 48
on-line documentation, xxiii
OPEN specifier FILEOPT, 93
opt/SUNWspro standard location for Sun software, 42, 120
optimization
files, 234
global, 4
inline user-written routines, 40
object code, 49
peephole, 4
performance, 36
performance tuning, 234
splitting, 234
optimizer out of memory, 71
option
debugging, useful, 153
differences for Solaris 2.x, 1.x, x86, 7
frequently used options, 25
list, 40
pass to program, 52
OPTIONS, 69
options, 25
listed by option name, 29
listed by what they do, 26
most frequently used, 25
show list of, -help, 40
order of
functions, 58
linker search, 120, 121
options on command line, -lx, 119
original case, 57
output
file, naming it, 48
from an exception handler, 11
redirection, 82
standard, 89
to terminal, -Xlist, 142

overflow
IEEE, 175
stack, 55
with reductions, 322
overriding macro values, 104

P

-p, profile by procedure, 50
page-align common blocks, 30, 70
PARALLEL, number of processors, 307
-parallel, parallelize loops, 50
parallelization
automatic, 31, 313
CALL in a loop, 55
debug tips, 309
explicit, 35, 325
general requirements, 303
loop information, 42
number of processors, 308
overview, 304
reduction, 53
speed gained or lost, 306, 307
summary table, 305
warnings, 57
parts of large arrays in dbx, 165
pass
arguments by reference, 241
arguments by value, 241
file names to programs, 88
option to program, 52
passes of the compiler, 57
path, 78
#include, 41
library search, 117
name, 80
absolute, 80
complete, 80
relative, 80
peephole optimization, 4
-pentium, 50
performance
case history, 230
lessons, 234
optimization, 36
time command, 230
tuning and optimization, 234
-pg, profile by procedure, 51
-PIC, 51
-pic, 126
-pic, 51
piping, 83
standard error, 170
pixrect with XView, 290
porting, 203
carriage-control, 207
file-equates, 208
formats, 207
guidelines, 213
problems, checking, -Xlist, 141
position-independent code, 51
and -pic, 126
POSIX
bindings, 135, 136
documents, 136
option, 135
runtime checking, 135
pragma, 67
C() directive, 240
explicit parallelization, 67, 68, 327
general, 67
parallel, 68
preattached
files, 90
logical units, 90
preconnect units 0, 5, 6 from C, 280
preconnected units, 89
preprocessor, 23
prerequisites, xxii
preserve case, 57, 240

print
array
parts of large, in dbx, 165
slices in dbx, 165
asa, 15
procedure
names, 240
profile -pg gprof, 51
process control, dbx, 167
processors, number for
parallelization, 307
produce
position-independent code, 51
source code of specified type, 52
prof, -p, 50
profile
gprof, 15, 218
I/O, 223
libraries missing, 224
tcov, 15, 221
time, 216
profile by
basic block, 30
loop for MP, -Zlp, looptool, 62
procedure, -p, prof, 50
procedure, -pg, gprof, 51
prompt
conventions, xxvi
only, 55
pstat, actual swap space, 1.x, 73
pure scalar variable, 315
purpose of manual, xxi
pwd, 79

Q

-Qoption, 52
–Qpath, 52
-Qproduce, 52
quadruple precision trigonometric
functions, 12

R

-R and LD_RUN_PATH, not identical, 54
-R list, 54
-r option for ar, 125
-r8, 53
ran, 135
random I/O, 92
range of subscripts, 32
ranlib, randomize static library, 123
ratfor read me file, xxv
READMEs, xxv
reads, number of, 217
REAL as DOUBLE PRECISION, 53
recursive I/O, 11, 45
redirection, 82
standard error, 83, 170
-reduction, parallelize automatically,
with reduction, 53
reductions
for automatic parallelization, 320
recognized by the compiler, 321
roundoff with automatic
parallelization, 322
reference
versus value, C/FORTRAN, 241
referenced but not declared, checking,
-Xlist, 141
relative path name, 80
remove .fln files, 143
rename executable file, 18
reorder functions, 58
replace library module, 125
retrospective of accrued exceptions, 194
return function values to C, 273
risk with explicit parallelization, 333
root, 77
roundoff with reductions, 322
run path in executable, 47
running FORTRAN, 18
runtime error messages, 283

S

-S, 56
-s, 55
safe libraries for multi-thread programming, 44
sample interface C FORTRAN, 235
-sb, source browser, 55
-sbfast, 55
sbrowser debug, 15
SCCS, 105
  checking in files, 111
  checking out files, 111
  creating files, 107
  inserting keywords, 106
  making directory, 106
  putting files under SCCS, 106
search
  object library directories, 43
  order for libraries, 120
  path compilation directory, 52
secnds, 135
segmentation fault, 32, 55, 153, 160
  some causes, 159
  use -C to find line number, 160
  use dbx to find line number, 160
set
  #include path, 41
  directory for
    .fln files, 149
    temporary files, 56
  LD_LIBRARY_PATH, 118
  level of checking strictness, -Xlist, 150
  nesting level of
    control structures, 47
    data structures, 47
set (*continued*)
  number of
    continuation lines, 47
    equivalenced variables, 48
    external names, 48
    identifiers, 47
    processors for parallelization, 307
    statement numbers, 48
  search path compilation directory, 52
Shakespeare, 235
shared library, 125
  build, -G, 39
  name a shared library, 39
sharing I/O C FORTRAN, 265, 280
shell
  limits, 72
  script, 99
shorten command lines
  alias, 69
  environment variable, 69
show commands, 34
SIG, Sun Programmers Special Interest Group, xxvi, 357
SIGBUS
  some causes, 162
SIGFPE, 38
  definition, 176, 184
  generate, 184
  when generated, 186, 195
signal
  handler, 186
  with explicit parallelization, 335
SIGSEGV
  changing a constant, 9
  some causes, 159
-silent, 55
size
  four-byte integers, 40
  of data types, 239
slices of arrays in dbx, 165
slower loading, 9
Solaris, 2

source
browser, 55
catalogs, 64
code, produce specified type, 52
diagnostics, 168
lines -e, 35
SourceBrowser, 55
speed gained or lost from parallelization, 306
spirits, 235
splitting and optimization, 234
stack
overflow, 55
variables, 55
stack trace, 163
-stackvar, 55
standard
arithmetic, 194
conformance to standards, 3
error, 83
redirecting in csh() and sh(), 170
error, accrued exceptions, 194
input, 82, 89
output, 82, 89
statement
numbers, number of, 48
profile by, -a and tcov, 30
unreachable, checking, -Xlist, 141
static
binding, 34
library, 122
tables in compiler, 47
strictness of checking, -Xlist, 150
strip executable of symbol table, -s, 55
strong typing, 57
structure alignment, 48
stupid UNIX tricks
shorten command line, alias, 69
shorten command line, environment variable, 69
subprogram in loop, explicit parallelization, 329
subroutine
compared to function, C FORTRAN, 238
names, 240
unused, checking, -Xlist, 141
used as a function, checking, -Xlist, 141
subscripts check, 32, 153, 160
with -C, 10
suffix
of file names recognized by compiler, 24
rules in make, 105
summing and reduction, automatic parallelization, 320
Sun Programmer Quarterly Newsletter, 357
Sun, sending feedback to, xxv, 40
SunOS
4.1.X, 2
5.x, 2
suppress
autoload for dbx, 61
blank in listed-directed output, 48
converting uppercase letters to lowercase, 57
display of entry names and file names, 55
error nnn, -Xlist, 148
implicit typing, 57
license queue, 46
linking, 32
unreferenced identifiers, -Xlist, 150
warnings
f77 warnings, 58
-Xlist, 151
SVR4, 2
swap, 72
swap space
display actual swap space, 72, 73
limit amount of disk swap space, 71
swapouts, number of, 217

symbol table
for dbx, 39, 61
strip executable of, 55
syntax
compiler, 22
errors, -Xlist, 141
f77, 22
system time, 216
System V Release 4 (SVR4), 2

T

tape
file representation, 96
multifile access, 98
tcov, 221
-a, profile by statement, 30
profile, 15
-temp, 56
templates inline, 42, 43
temporary files, directory for, 56
terminal display, -Xlist, 142, 150
textedit, 14
third-party software and hardware, 14
thread analyzer, 63
thread stack, 56
time
compilation passes, 57
execution, optimization, 49
functions, 204
system, user, etc., 216
time, 135
-time, 57
tips and hints, debug parallelized code, 309
traceback
dbx, 163
ode, 163
transporting, 203
carriage-control, 207
file-equates, 208
formats, 207
tree, 77
triangle as blank space, xxvi
turn off warnings about IEEE accrued exceptions, 180
type checking across routines, -Xlist, 141
typewriter font, xxvi
typing, strong, 57

U

-u, 57
-U do not convert to lowercase, 57, 240
UCB 4.3 BSD, 2
ulimit, 72
undeclared
default type, 57
variables, 153
underflow
abrupt, 194
forced to zero, 37
gradual, 194
IEEE, 175
with reductions, 322
underscore
do not append to external names, 67
external names with, 67
in external names, 241
unformatted record size, 59
unit
logical unit preattached, 90
preconnected units, 89
unrecognized options, 24
unrequited exceptions, 194
unresolved reference, order on command line, -lx, 119
-unroll, unroll loops, 57
unused functions, subroutines, variables, labels, -Xlist, 141
upgrading from
1.4, 12
2.0/2.0.1, 9
3.0, 8

uppercase
debug, 166
external names, 240
usage
automatic parallelization, 313
compiler, 22
explicit parallelization, 325
TOPEN, 95
user time, 216

V

-V, 58, 153, 154
-v, 57
VAL(), pass by value, 241
variable
unused, checking, -Xlist, 141
used but unset, checking, -Xlist, 141
vasty deep, 235
verify agreement across routines, -Xlist, 139
version
checking, 154
id of each compiler pass, 58
vi, 14
VMS
debug statements, d, 60
features with -xl, 59
library, 135
routines, 135

W

-w, 58
warnings
explicit parallelization, 330
suppress f77 warnings, 58
watchpoints, dbx, 167
where
exception occurred, 196
exception occurred, by line number, 161
execution stopped, 163
width of output lines, -Xlist, 151
wimp
interface SourceBrowser, 15
interface to dbx, 15, 167
writes, number of, 217

X

X Windows, 289
X11 interface, 135
X3.9-1978, 3
-xa, 58
-xcgyear, 58
xemacs, 14
-xF, 58
-xinline, 59
-xl, extended language, VMS, 59
-xld, 60, 155
-xlibmil, 60
-xlicinfo, 60
-Xlist, 142
a la carte options, 147
combination special, 147
defaults, 142
display directly to terminal, 142
errors and
call graph, -Xlistc, 147
cross reference, -XlistX, 147
listing, -XlistL, 147
sample usage, 143
suboptions, 147
details, 148
summary, 148
-Xlist, global program checking, 62, 139
-Xlistc, 148
-XlistE, 147, 148
-Xlisterr, 148
-Xlistf, 149
-Xlistfln*dir*, 143
.fln files directory, 149
-Xlists, 150
-Xlistv*n*, 150
-Xlistw, 151

-Xlistwar, 151
-XlistX, 151
-xnolib, 60, 61
-xpg, 61
-xs, disable autoload, 61
-xsb, 61
-xsbfast, 61
XView, 290, 294, 296
    Toolkit, 289
    translate C to FORTRAN, 298

Z

zero
    division by, 174, 175
    on underflow, 37
-Zlp, loop profiler, MP, 62
-Ztha, prepare for thread analyzer, 63

Join the SunPro SIG Today

Sun Programmer Special Interest Group

The benefits are SIGnificant

At SunSoft, in the Software Development Products business of Sun Microsystems, our goal is to meet the needs of professional software developers by providing the most useful line of software development products for the Solaris platform. We've also recently formed a special interest group, SunPro SIG, designed to provide a worldwide forum for exchanging software development information. This is your invitation to join our world-class organization for professional programmers. For a nominal annual fee of $20, your SunPro SIG membership automatically entitles you to:

- Membership on an International SunPro SIG Email Alias

  Share tips on performance tuning, product feedback, or anything you wish; available as a UUNET address and a dial-up number

- Subscription to the SunProgrammer Quarterly Newsletter

  Includes advice on getting the most out of your code, regular features, guest columns, product previews and the latest industry gossip

- Access to a Repository of Free Software

  SunSoft will collect software related to application development and make it available for downloading

- Free SunSoft Best-of-Repository CD-ROM

  Periodically, we'll take the cream of the crop from the depository and distribute it to members annually

- Free Access to SIG Events

  Including national events, like SIG seminars held at the SUG conference, and regional SunPro SIG seminars

SPECIAL OFFER

Sign up today, and receive a SunPro SIG Tote Bag

A spiffy 15" x 12" black nylon Cordura tote with the SIG logo proof positive of your Power Programmer status

So join the SunPro SIG today. And plug into what's happening in SPARC and Solaris development world-wide. Simply complete the form below.

Mail to: SunPro SIG, 2550 Garcia Avenue MS UMPK 03-205, Mountain View, CA,94043-1100

TEL: (415) 688-9862

or

FAX: (415) 968-6396

Unfortunately we cannot accept credit card orders via Email since we need to have your signature on file.

Sign me up for SunPro SIG!

Sun Programmer Special Interest Group

Date	
Name	
Title	
Company	
Email Address	
Address	
City	State
ZIP	Country
Phone	
Fax	

ALL INFO MUST BE FILLED OUT

SunSoft, A Sun Microsystems, Inc. Business

I'd like to pay for my one-year membership fee of $20 by:

❒ VISA

❒ MASTERCARD

Card # _____

Expiration Date: _____

Signature:

_____

❒ Check made payable to SunSoft