

Fortran 90 User's Guide

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

Part No.: 801-5492-10
Revision A, March 1995

© 1995 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. CRAY is a registered trademark of Cray Research, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PRODUCT IS DERIVED FROM CRAY CF90™, A PRODUCT OF CRAY RESEARCH, INC.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Contents

Preface	xvii
1. Introduction.....	1
1.1 Operating Environment.....	2
1.2 Text Editing.....	2
1.3 Program Development.....	3
1.4 Debugging.....	3
1.5 Licensing	3
2. Getting Started	5
2.1 Summary	5
2.2 Compiling	6
2.3 Running.....	6
2.4 Renaming the Executables	6

3. Using the Compiler	9
3.1 Compile Command	9
Purpose	10
Compile Link Sequence	10
Command-line File Names	11
Unrecognized Arguments	12
3.2 Compiler Options.....	12
Actions/Options Frequently Used.....	13
Actions and What Options Invoke Them	13
Options and What Actions They Do	15
3.3 Miscellaneous Tips.....	33
Floating-Point Hardware Type	33
Many Options on Short Commands.....	33
4. File System and File I/O	35
4.1 Summary.....	35
4.2 Directories.....	37
4.3 File Names.....	37
4.4 Path Names.....	37
Relative Path Names	38
Absolute Path Names.....	38
4.5 Redirection	40
4.6 Piping.....	41

4.7 Accessing Files from Fortran Programs.....	42
Accessing Named Files	42
Accessing Unnamed Files	43
Passing File Names to Programs	43
4.8 Direct I/O	45
4.9 Internal Files	46
5. Program Development.....	49
5.1 Simple Program Builds	49
Writing a Script.....	49
Creating an Alias	50
Using a Script or Alias	50
Limitations.....	50
5.2 Program Builds with the make Program.....	50
The make File	50
Using make	51
5.3 Tracking and Controlling Changes with SCCS.....	52
Putting Files under SCCS.....	52
Checking Files Out and In.....	54
6. Libraries	57
6.1 Libraries in General	57
Load Map.....	58
Advantages of Libraries.....	58

6.2 Static Libraries	58
Disadvantages of Libraries	58
Sample Creation of a Static Library	59
Sample Replacement in a Static Library	61
6.3 Dynamic Libraries	61
Performance Issues.....	62
Binding Options	62
A Simple Dynamic Shared Library.....	63
6.4 Consistent Compile and Link.....	65
6.5 Library Paths.....	65
Installation Directory	65
Building Executables: ld Search order.....	66
Running Executables: ld Search order	66
Build Paths and Run Paths	67
Finding Built-in Paths	67
7. Debugging	69
7.1 Global Program Checking (-xlist)	69
Errors in General.....	70
Details	70
Using Global Program Checking	71
Suboptions for Global Checking Across Routines	73

7.2 The dbx Debugger	77
Sample Program for Debugging	78
A Sample dbx Session	79
Segmentation Fault—Finding the Line Number	81
Exception—Finding the Line Number	83
Trace of Calls	84
Pointer to a Scalar	85
Pointer to an Array	86
User-Defined Types	87
Pointer to User-Defined Type	89
Allocated Arrays	91
Print Arrays	92
Print Array Slices	93
Generic Functions	94
Miscellaneous Tips	96
Main Features of the Debugger	96
Help	97
8. Floating Point	99
8.1 Summary	99
8.2 The General Problems	100
8.3 IEEE Solutions	101

8.4 IEEE Exceptions	102
Detecting a Floating-point Exception.	102
Generating a Signal for a Floating-point Exception	102
Default Signal Handlers.	102
8.5 IEEE Routines	103
Flags and <code>ieee_flags()</code>	104
Values and <code>ieee_values()</code>	106
Exception Handlers and <code>ieee_handler()</code>	107
Retrospective.	111
Nonstandard Arithmetic	111
Messages about Floating-point Exceptions	112
8.6 Debugging IEEE Exceptions	113
8.7 Guidelines	113
8.8 Miscellaneous Examples	114
Kinds of Problems.	114
Simple Underflow.	115
Use Wrong Answer.	116
Excessive Underflow	116
9. C–Fortran Interface	119
9.1 Sample Interface	120
9.2 How to Use this Chapter	121

9.3 Compatibility Requirements	122
Function or Subroutine	123
Underscore in Names of Routines	124
Case Sensitivity.....	124
Data Type Compatibility	125
Passing Arguments by Reference or Value	127
Character Strings and Order	129
Array Indexing and Order.....	130
Libraries and Linking with the <code>f90</code> Command	131
File Descriptors and <code>stdio</code>	132
File Permissions	133
9.4 Fortran Calls C	134
Arguments Passed by Reference (<code>f90</code> Calls C).....	134
Arguments Passed by Value (<code>f90</code> Calls C)	141
Function Return Values (<code>f90</code> Calls C)	141
Labeled Common (<code>f90</code> Calls C)	147
Alternate Returns (<code>f90</code> Calls C) - N/A	148
9.5 C Calls Fortran	149
Arguments Passed by Reference (C Calls <code>f90</code>).....	149
Arguments Passed by Value (C Calls <code>f90</code>) - N/A	155
Function Return Values (C Calls <code>f90</code>)	155
Labeled Common (C Calls <code>f90</code>)	160
Alternate Returns (C Calls <code>f90</code>)	161

A. Features and Differences	163
A.1 Standards.....	163
A.2 Extensions.....	164
Tabs in the Source.....	164
Continuation Line Limits.....	165
Fixed-Form Source of 96 Characters.....	165
Directives.....	165
Source Form Assumed.....	165
Boolean Type.....	167
Abbreviated Size Notation for Numeric Data Types.....	170
Cray Pointers.....	171
Cray Character Pointers.....	176
Intrinsics.....	178
A.3 Directives.....	179
General Directives.....	179
Form of General Directive Lines.....	180
FIXED and FREE Directives.....	181
Parallel Directives.....	182
Form of Parallel Directive Lines.....	182

A.4 Compatibility with FORTRAN 77	184
Source	184
Executables	184
Libraries	184
I/O	185
Intrinsics	187
A.5 Forward Compatibility	188
A.6 Mixing Languages	188
A.7 Module Files	188
B. iMPact: Multiple Processors	189
B.1 Requirements	189
B.2 Overview	190
Automatic Parallelization	190
Explicit Parallelizing	190
Summary	191
Standards	191
B.3 Speed Gained or Lost	192
B.4 Number of Processors	192
C. iMPact: Automatic Parallelization	195
C.1 What You Do	195
C.2 What the Compiler Does	196
Parallelize the Loop	196
Dependency Analysis	196
Definitions: Array, Scalar, and Pure Scalar	197

C.3 Definition: Automatic Parallelizing	197
General Definition	197
Details	197
Exceptions for Automatic Parallelizing	198
D. iMPact: Explicit Parallelization	201
D.1 What You Do	201
D.2 What the Compiler Does	202
D.3 Parallel Directives	203
Form of Directive Lines	203
DOALL Parameters	205
D.4 DOALL Loops	206
Definition	206
Explicitly Parallelizing a DOALL Loop	206
CALL in a Loop	208
D.5 Exceptions for Explicit Parallelizing	208
D.6 Risk with Explicit: Nondeterministic Results	209
Testing is not Enough	209
How Indeterminacy Arises	210
D.7 Signals	211
Index	213
Join the SunPro SIG Today	227

Figures

Figure 4-1	File System Hierarchy	36
Figure 4-2	Relative Path Name	38
Figure 4-3	Absolute Path Name	39

Tables

Table 3-1	File Name Suffixes Fortran 90 Recognizes.	11
Table 3-2	Options Frequently Used	13
Table 3-3	Actions/Options Sorted by Action.	13
Table 3-4	Summary of -Xlist Suboptions.	32
Table 7-1	-Xlist Combination Special or A La Carte Suboptions	74
Table 7-2	-Xlist Suboptions Summary	74
Table 8-1	ieee_flags Argument Meanings	105
Table 8-2	Functions for Using IEEE Values	107
Table 9-1	C Data Type to Fortran 90 Data Type	125
Table 9-2	Fortran 90 Data Type to C Data Type	126
Table 9-3	Characteristics of Three I/O Systems.	132
Table A-1	Size Notation for Numeric Data Types	170
Table A-2	Nonstandard Intrinsics.	178
Table A-3	General Directives Guaranteed Only in the Current Release.	179
Table A-4	Parallel Directives Guaranteed Only in the Current Release .	182

Table B-1	Parallelization Summary	191
Table D-1	DOALL General Parameters	205
Table D-2	DOALL Scheduling Parameters	205

Preface

This preface is organized into the following sections.

<i>Purpose and Audience</i>	<i>page xvii</i>
<i>Before You Read This Book</i>	<i>page xviii</i>
<i>How This Book is Organized</i>	<i>page xviii</i>
<i>Related Documentation</i>	<i>page xviii</i>
<i>Conventions in Text</i>	<i>page xxii</i>

Purpose and Audience

This guide shows how to use Sun Fortran 90 1.0. Major topics of the guide are:

- Using the compiler command and options
- Global program checking across routines
- Using iMPact™ multiprocessor Fortran 90 MP
- Making and using libraries
- Using some utilities and development tools
- Using IEEE floating point with Fortran 90
- Using debuggers with Fortran 90
- Mixing C and Fortran 90

The guide is intended for scientists and engineers with the following:

- Thorough knowledge of Fortran 90
- General knowledge of some operating system (experience with some OS)
- Particular knowledge of the SunOS™ commands `cd`, `pwd`, `ls`, `cat`

Before You Read This Book

If you are not familiar with Fortran 90, you may want to consult the following.

- *Fortran 90 Handbook* (Fortran 90 language definition, including intrinsics)
- *Fortran 90 Explained* (Text book introduction to Fortran 90)

See “Related Manuals” on page xix.

How This Book is Organized

This book is organized as follows.

<i>Chapter 1, Introduction</i>	<i>page 1</i>
<i>Chapter 2, Getting Started</i>	<i>page 5</i>
<i>Chapter 3, Using the Compiler</i>	<i>page 9</i>
<i>Chapter 4, File System and File I/O</i>	<i>page 35</i>
<i>Chapter 5, Program Development</i>	<i>page 49</i>
<i>Chapter 6, Libraries</i>	<i>page 57</i>
<i>Chapter 7, Debugging</i>	<i>page 69</i>
<i>Chapter 8, Floating Point</i>	<i>page 99</i>
<i>Chapter 9, C–Fortran Interface</i>	<i>page 119</i>
<i>Appendix A, Features and Differences</i>	<i>page 163</i>
<i>Appendix B, iMPact: Multiple Processors</i>	<i>page 189</i>
<i>Appendix C, iMPact: Automatic Parallelization</i>	<i>page 195</i>
<i>Appendix D, iMPact: Explicit Parallelization</i>	<i>page 201</i>

Related Documentation

The related kinds of documentation included with Fortran 90 are as follow:

- Paper manuals (hard copy)
- On-line manuals in the AnswerBook™ viewing system
- On-line man pages
- f90 -help variations
- On-line READMEs directory of information files

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 manual *Installing SunSoft Developer Products on Solaris*.

Related Manuals

The following manuals are provided on-line or on paper, as indicated.

Title	Part Number	Paper	AnswerBook
<i>Fortran 90 User's Guide</i>	801-5492-10	X	X
<i>Fortran 90 Handbook</i> , by Adams, Brainerd, et al	875-1202-10		X
<i>Fortran 90 Browser</i>	802-2190-10	X	X
<i>Debugging a Program</i>	801-7105-10	X	X
<i>Numerical Computation Guide</i>	801-7639-10	X	X
<i>Installing SunSoft Developer Products on Solaris</i>	802-1561-10	X	X
<i>What Every Computer Scientist Should Know About Floating-Point Arithmetic</i>	800-7895-10		X

man *Pages*

Purpose

A man page is intended to answer “What does it do?” and “How do I use it?”

- *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 major highlights. It is a *quick* reference, not a *complete* reference.

Using man Pages

To display a man page, use the `man` command.

Example: Display the `f90` man page.

```
demo$ man f90
```

Example: Display the man page for the `man` command.

```
demo$ man man
```

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

Related man Pages

The following man pages may be of interest to Fortran 90 users.

<code>f90(1)</code>	Invoke the Fortran 90 compiler.
<code>asa(1)</code>	Print files having Fortran carriage-control.
<code>dbx(1)</code>	Debug by a command-line-driven debugger.
<code>debugger(1)</code>	Debug by a graphical-user-interface .
<code>fpr(1)</code>	Print files having Fortran carriage-control.
<code>ieee_flags(3M)</code>	Examine, set, or clear floating-point exception bits.
<code>ieee_handler(3M)</code>	Handle exceptions.
<code>matherr(3M)</code>	Handle errors.

f90 -help Variations

The following variations are meant to suggest other possibilities.

f90 -help more	The list does <i>not</i> scroll off the screen.
f90 -help grep "par"	Show only parallel options.
f90 -help grep "lib"	Show only library options.
f90 -help lp	Print a copy on paper.
f90 -help > MyWay	Put list onto a file, regroup, reorder, delete, ...
f90 -help tail	Show how to send feedback to Sun.

READMEs

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

	Standard Installation	Nonstandard Installation to <i>/my/dir/</i>
Location	/opt/SUNWspro/READMEs/	<i>/my/dir/</i> SUNWspro/READMEs/

	Contents
File Names	
feedback	Sun programmers email template file: Send feedback comments to Sun
fortran_90	Fortran 90 bugs, new features, behavior changes, documentation errata

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 215.

Conventions in Text

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

- We show code listing examples in boxes.

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

- Plain typewriter font shows prompts and coding.
- In dialogs, **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.
- A small clear triangle Δ shows a blank space where that is significant.

```
 $\Delta\Delta$ 36.001
```

- 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 X3.198-1992 standard, as described in *American National Standard for Programming Language—Fortran—Extended*, ANSI X3.198-1992, 1992, American National Standards Institute, Inc., informally abbreviated as the Fortran 90 Standard.
- We usually show Fortran 90 examples in free form, not fixed form or tab.
- We usually abbreviate “Sun Fortran 90” as “f90”.
- We usually show Fortran 90 keywords and intrinsics in uppercase, and all else in lowercase or mixed case.

Introduction



This chapter is organized into the following sections.

<i>Operating Environment</i>	<i>page 2</i>
<i>Text Editing</i>	<i>page 2</i>
<i>Program Development</i>	<i>page 3</i>
<i>Debugging</i>	<i>page 3</i>
<i>Licensing</i>	<i>page 3</i>

Sun Fortran 90 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 TeamWare™. 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 90 Workshop
- With or without the iMPact MT/MP multiple processor package

1.1 Operating Environment

Sun Fortran 90 runs in the Solaris® 2.x operating environments.

The Solaris 2.x operating environment includes (among other components) the SunOS™ 5.x operating system. 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™.

Abbreviations

- Solaris 2.x is an abbreviation for “Solaris 2.3 and later.”
- SunOS 5.x is an abbreviation for “SunOS 5.3 and later.”

1.2 Text Editing

There are several text editors available.

vi The major text editor for source programs is `vi` (vee-eye), the visual display editor. It has considerable power because it offers the capabilities of both a line and a screen editor. `vi` also provides several commands specifically for editing programs. These are options you can set in the editor. Two examples are the `autoindent` option, which supplies white space at the beginning of a line, and the `showmatch` option, which shows matching parentheses. For more information, read the `vi` section of the manual.

textedit The `textedit` editor and other editors are available, including `ed` and `ex`.

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`

1.3 Program Development

There are several development tools available.

`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)` and `fpr(1)`.

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

1.4 Debugging

There are two debugging tools.

`dbx` An interactive symbolic debugger that understands Sun Fortran 90 programs (available with the SPARCworks set).

`debugger` A window, icon, mouse, and pointer interface to `dbx` (in SPARCworks set).

1.5 Licensing

This compiler uses network licensing. Before you use Sun Fortran 90, purchase and install a SunSoft Fortran 90 license.

When you invoke the compiler, if a license is available, the compiler simply 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.

See also `-noqueue` and `-xlicinfo`.

Licensing information is in the manual *Installing SunSoft Developer Products on Solaris*, including (among other items):

- Installing a license
- Starting a license daemon
- Restarting a license daemon after a license server crash

Getting Started



This chapter is organized into the following sections.

<i>Summary</i>	<i>page 5</i>
<i>Compiling</i>	<i>page 6</i>
<i>Running</i>	<i>page 6</i>
<i>Renaming the Executables</i>	<i>page 6</i>

This chapter gives a bare minimum on how to compile and run Fortran 90 programs under Solaris. This chapter is for you if you know Fortran 90 thoroughly and need to start writing programs in this Fortran 90 immediately. Skip to Chapter 3, “Using the Compiler,” to learn more about it first.

2.1 Summary

Before you use this release of `f90`, it must be installed and licensed. Read *Installing SunSoft Developer Products on Solaris*

To use this Fortran 90 involves three steps:

- Write and save a Fortran 90 program; use `.f90` or `.f` as file name suffix.
- Compile and link this file using the `f90` command.
- Execute by typing the name of the executable file.

Example: This program displays a message on the screen.

```
demo$ cat hack.f90
PROGRAM Opinion
  PRINT *, 'Real programmers hack Fortran 90!'
END PROGRAM Opinion
demo$ █
```

2.2 Compiling

Compile and link using the `f90` command as follows.

```
demo$ f90 hack
demo$ █
```

In the example above, `f90` compiles `hack.f90` and puts the executable code in 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 90!
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 maven
demo$ █
```

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

```
demo$ f90 -o maven hack
demo$ █
```

The above command places the executable code in the `maven` file.

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

```
demo$ maven  
Real programmers hack Fortran 90!  
demo$ █
```

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

Using the Compiler



This chapter is organized into the following sections.

<i>Compile Command</i>	<i>page 9</i>
<i>Compiler Options</i>	<i>page 12</i>
<i>Miscellaneous Tips</i>	<i>page 33</i>

3.1 Compile Command

Before you use this release of f90, it must be installed and licensed. Read ***Installing SunSoft Developer Products on Solaris***

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

```
f90 [options] sfn ...
```

where *sfn* is a Fortran 90 source file name, and *options* is one or more of the compiler options.

Example: A compile command with two files.

```
demo$ f90 growth.f90 fft.f90
```

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

```
demo$ f90 -g -P growth.f90 fft.f90
```

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

```
f90 [options] fn ... [-lx]
```

- The *fn* is a file name (not necessarily of a Fortran 90 source file). See “Command-line File Names” on page 11.
- The *-lx* is the option to link with library *libx.a*.
- The *-lx* 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.

Purpose

The purpose of *f90* is to translate source to an executable file.

Other major uses:

- Translate source code files to relocatable binary (*.o*) files
- Link *.o* files into an executable load module (*a.out*) file
- Show the commands built by the compiler, but do not execute
- Prepare for debugging

Compile Link Sequence

With the above commands, if you successfully compile the files *growth.f90* and *fft.f90*, 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.

Compile and Link in Separate Steps

You can compile and link in separate steps. This is usually done if one of several source files was changed—that way you need not recompile all the other source files.

Example: Compile and link in separate steps.

```
demo$ f90 -c file1.f90 file2.f90 file3.f90 {make .o files}
demo$ f90 file1.o file2.o file3.o {make a.out file}
```

Of course, every file named in the first step (as a `.f90` file) must also be named in the second step (as a `.o` file).

Consistent Compile and Link

Be consistent with compiling and linking. If you compile and link in separate steps, and you *compile* any subprogram with `-dalign` or `-fast`, then be sure to *link* with the same options.

Command-line File Names

If a file name in the command line has any of the following suffixes, the compiler recognizes it; otherwise it is passed to the linker.

Table 3-1 File Name Suffixes Fortran 90 Recognizes

Suffix	Language	Form	Action
<code>.f90</code>	Fortran 90	Free	Compile Fortran 90 source files, put object files in current directory; default name of object file is that of the source but with <code>.o</code> suffix.
<code>.f</code>	Fortran 90 or standard FORTRAN 77	Fixed	Same as <code>.f90</code> , but different source form
<code>.for</code>	Same as <code>.f</code>	Fixed	Same as <code>.f</code>
<code>.ftn</code>	Same as <code>.f</code>	Fixed	Same as <code>.f</code>
<code>.s</code>	Assembler		Assemble source files with the assembler.
<code>.o</code>	Object Files		Pass object files through to the linker.

Fixed-form source and free-form source are explained in Section 3.3 Source Form, of the *Fortran 90 Handbook*.

Unrecognized Arguments

Any arguments `f90` does not recognize are taken to be one of the following:

- Linker option arguments
- Names of `f90`-compatible object programs (maybe from a previous run)
- Libraries of `f90`-compatible routines

If an unrecognized argument:

- Has a “-”, then it is an *option*, and generates a warning.
- Has no “-”, then it generates no *warnings*; but if the linker does not recognize them, the linker issues *error* messages.

3.2 Compiler Options

This compiler has the power of *many* optional features, so this tends to produce a very long list of features. To help you use this long list, the options are organized from different perspectives—so you can look up an action to see which option does it, or you can look up an option to see what it does.

<i>List/Perspective</i>	<i>How to Get It</i>
<ul style="list-style-type: none"> ● Actions and What Invokes Them (<i>Actions/Options Sorted by Action</i>) <ul style="list-style-type: none"> Frequent—Actions/Options Frequently Used <i>See page 13</i> Summary—One-line descriptions <i>See page 13</i> (See also “compile action” in the Index.) ● Options and What They Do (<i>Options/Actions Sorted by Option</i>) <ul style="list-style-type: none"> Summary—One-line descriptions <code>f90 -help</code> Full Descriptions—Examples, risks, trade-offs, restrictions, interactions <ul style="list-style-type: none"> <i>All</i> risks, trade-offs, restrictions, interactions, and examples <i>See page 15, ...</i> <i>Some</i> risks, trade-offs, restrictions, interactions, examples <code>man f90</code> (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—produce additional symbol table information for the debugger.	-g	page 20
Performance—make executable run faster using a selection of options.	-fast	page 19
Performance—make executable run faster using the optimizer.	-O[n]	page 25
Bind as dynamic (or static) any library listed later in the command. -Bdynamic, -Bstatic	-Bbinding	page 15
Library—Allow or disallow dynamic libraries for the entire executable. -dy, -dn	-dbinding	page 17
Compile only—suppress linking; make a .o file for each source file.	-c	page 16
Name the final output file <i>nm</i> instead of <i>a.out</i> .	-o nm	page 25
Display a list of compiler options.	-help	page 20

Check “*Details*” for trade-offs, risks, restrictions, interactions, and examples.

Actions and What Options Invoke Them

Actions/Options Sorted by Action—This section groups related actions together. Check “*Details*” for risks, trade-offs, restrictions, interactions, and examples.

Table 3-3 Actions/Options Sorted by Action

Action	Option	Details
Debug		
Compile for use with the debugger.	-g	page 20
Global checking—across routines (arguments, commons, parameters, ...).	-xlist	page 32
Version ID—show version ID along with name of each compiler pass.	-V	page 30
Library		
Bind as dynamic (or static) any library listed later in the command.	-Bbinding	page 15
Allow or disallow dynamic libraries for the entire executable.	-dbinding	page 17
Build a dynamic shared library.	-G	page 20
Name a shared dynamic library.	-hname	page 20
Directory—search this directory first.	-Ldir	page 22
Link with library <i>libx</i> .	-lx	page 21

Table 3-3 Actions/Options Sorted by Action (Continued)

Action	Option	Details
Library (continued)		
Multithread safe libraries, low level threads.	-mt	page 22
Paths—store into object file.	-R <i>list</i>	page 28
No automatic libraries.	-nolib	page 24
No run path in executable.	-norunpath	page 24
Performance		
Faster execution—make executable run faster using a selection of options.	-fast	page 19
Generate code to run on generic SPARC architecture.	-cg89	page 16
Generate code to run on SPARC V8 architecture.	-cg92	page 16
Use the best floating-point arithmetic for this machine.	-native	page 23
Optimize for execution time.	-O[n]	page 25
Use selected math routines optimized for performance.	-xlibmopt	page 31
Reset -fast so that it does not use -xlibmopt.	-xnolibmopt	page 30
Parallelization		
Parallelize automatically and with explicit directives.	-parallel	page 26
Parallelize explicitly.	-explicitpar	page 18
Stack local variables to allow better optimizing with parallelizing.	-stackvar	page 29
Profile by		
Procedure for gprof.	-pg	page 27
Procedure for prof.	-p	page 26
Information and Warnings		
Verbose—print name of each compiler pass.	-v	page 30
Version ID—show version ID.	-V	page 30
Warnings—suppress warnings.	-w	page 31
Licensing		
License information—display license server user ids.	-xlicinfo	page 30
No license queue.	-noqueue	page 24
Source Forms		
Fixed form source.	-fixed	page 19
Free form source.	-free	page 19

Table 3-3 Actions/Options Sorted by Action (Continued)

Action	Option	Details
Miscellaneous		
ANSI conformance check—identify many non-ANSI extensions.	-ansi	page 15
Compile only, do not make a.out, do not execute	-c	page 16
CIF—generate a compiler information file.	-db	page 17
Command—show command line built by driver, but do not execute.	-dryrun	page 17
Align on 8-byte boundaries.	-f	page 19
Do not trap floating-point exceptions.	-fnonstop	page 19
Options—display the list of options.	-help	page 20
Include path—add <i>dir</i> to the search path for INCLUDE statements.	-I <i>dir</i>	page 21
Module directory—look for Fortran 90 modules in the <i>dir</i> directory.	-mdir	page 23
Output—rename the output file.	-o <i>outfil</i>	page 25
DO loops—use one trip DO loops.	-onetrip	page 25
Pass option list to program.	-Qoption <i>pr ls</i>	page 27
Assembly source—generate only assembly source code.	-S	page 30
Symbol table—strip executable of symbol table (prevents debugging).	-s	page 29
Temporary files—set directory to locate temporary files.	-temp= <i>dir</i>	page 30
Time for execution—display for each compilation pass.	-time	page 30

Options and What Actions They Do

Options/Actions Sorted by Option—This section shows all f90 options, with a full description, including risks, restrictions, caveats, interactions, examples, and other details.

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

-Bbinding Bind as dynamic (or static) any library listed later in the command.

No space is allowed between -B and dynamic or static, and either dynamic or static must be included.

- -Bdynamic: Prefer *dynamic* binding (try for shared libraries).
- -Bstatic: Require *static* binding (no shared libraries).

If you have neither -Bdynamic nor -Bstatic, you get the default: dynamic.

For `-Bdynamic` and `-Bstatic`, there is asymmetry besides *prefer/require*:

- If you specify *static*, but it finds only a dynamic version, then the library is not linked, and you get a warning that the “library was not found.”
- If you specify *dynamic*, but it finds only a static version, then the library is linked, and you get no warning.

You can toggle `-Bstatic` and `-Bdynamic` on the command line. That is, you can link some libraries statically and some dynamically by specifying `-Bstatic` and `-Bdynamic` any number of times on the command line.

These are loader/linker options. If you compile and link in separate steps, and you need `-Bbinding`, then you need it in the *link* step also.

-c Compile only, do not make a `.out`, do not execute.

Suppress linking by the loader. Make a `.o` file for each source file. Do not make an executable file. You can name a single object file explicitly using the `-o` option.

-cg89 Generate code to run on generic SPARC architecture.

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.

Allow the use of the full SPARC V8 instruction set.

General Comments on `-cg89` and `-cg92`

- 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.

- dalign** Allow `f90` to use double load/store.
- 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 90 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`.
- db** Compiler information file.
- Generate a compiler information (CIF) file.
- dryrun** Commands—show commands built by driver, but do not execute.
- d[y,n]** Allow or disallow *dynamic* libraries for the entire executable.
- No space is allowed between `-d` and `y` or `n`. Either `y` or `n` must be included.
- `-dy`: Yes—allow *dynamically* bound libraries (*allow* shared libraries).
 - `-dn`: No—do *not* allow dynamically bound libraries (*no* shared libraries).
- If you have neither `-dy` nor `-dn`, you get the default: `y`.
- These apply to the *whole* executable. Use only *once* on the command line.
- If `a.out` uses *only static* libraries, then `-dy` causes a few seconds delay at runtime it makes the *dynamic* linker be invoked when `a.out` is run. This takes a few seconds to invoke and find that no dynamic libraries are needed.
- `-dbinding` is a loader/linker option. If you compile and link in separate steps, and you need `-dbinding`, then you need it in the *link* step.
- e** Extend the source line maximum length to 132 columns.
- Accept lines up to 132 characters long.

-explicitpar Multiprocessor—parallelize explicitly.

You do the dependency analysis: analyze and specify loops for inter-iteration data dependencies. The software parallelizes the specified loops.

The `-explicitpar` option requires the Multiprocessor Fortran 90 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 B, “iMPact: Multiple Processors,” Appendix C, “iMPact: Automatic Parallelization,” and Appendix D, “iMPact: Explicit Parallelization.”

Summary: To parallelize explicitly, do the following.

- Analyze the loops to find those that are safe to parallelize.
- Insert `!MIC$ DOALL` to parallelize a loop.
- Use the `-explicitpar` option.

Example: Insert a parallel pragma immediately before the loop.

```
...
!MIC$ DOALL
  DO i = 1, n
    a(i) = b(i) * c(i)
  END DO
...
```

Example: Compile to explicitly parallelize.

```
demo$ f90 -explicitpar any.f90
```

Restrictions:

- `-g` turns off `-explicitpar`.
- Avoid `-explicitpar` if you do your own thread management. See `-mt`.
- Do not mix parallelized `f77` and parallelized `f90`.
- If you use `-explicitpar` and compile and link in *separate* steps, then *link* with `-explicitpar`.

- f** Align on 8-byte boundaries.
- 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.
- 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 the combination of compilation options that optimize for speed of execution without excessive compilation time.
- This should provide close to the maximum performance for most realistic applications.
- If you combine `-fast` with other options, the last specification applies.
- If you do not specify the level (as in `-fast -O`) you get `-fast -O3`.
- If you compile and link in separate steps, and you compile with `-fast`, then be sure to link with `-fast`.
- fixed** Fixed-form source.
- Interpret all Fortran 90 source files by fixed form rules. Overrides file suffix. See also, “FIXED and FREE Directives” on page 181.
- flags** Synonym for `-help`.
- fnonstop** Do not trap floating-point exceptions.
- Without `-fnonstop`, there is a trap on the invalid, overflow, and divide by zero floating-point exceptions.
- free** Free-form source.
- Interpret all Fortran 90 source files by free form rules. Overrides file suffix. See also “FIXED and FREE Directives” on page 181.

-g Debug—produce additional symbol table information for the debugger.

Produce a symbol table of information for the debuggers. You get much more debugging power if you compile with `-g` before using the debuggers.

- `-g` overrides `-O`.
- `-g` is turned off by `-explicitpar`, `-parallel`, or `-reduction`.

-G Library—build a dynamic library.

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

-hnm Library—make *nm* be the name of the generated shared dynamic library.

When generating a shared dynamic library, the compile-time linker records the specified name in the library file as the *internal* name of the library. If there is no `-hnm` option, then no internal name is recorded in the library file, and the *path* of the library is stored instead.

Advantage—If the library has an internal name, then whenever the executable is run, the linker needs only a library with the exact same internal name—it can be in *any path* the linker is searching. If the library has no internal name, then whenever the executable is run, the linker must find the library in the exact same *path* used when the executable was created. That is, the internal name method is more flexible.

Remarks

- A space between `-h` and *nm* is optional.
- In general, this name must be the same as what follows the `-o`.
- The `-hnm` option is meaningless without `-G`.
- Internal names facilitate *versions* of a dynamic library.
- This is a linker option.

See the *Linker and Libraries Manual*.

-help Options—display the list of options.

Display an equivalent of this list of options. This also shows how to send feedback comments to Sun.

Variations for `-help`:

- `f90 -help | more` Do not scroll the list off screen.
- `f90 -help | grep "par"` Show only parallel options.
- `f90 -help | tail` Show how to send feedback to Sun.

See also “`f90 -help Variations`” on page xxi.

`-Iloc` Include path—add to the search path for `INCLUDE` statements.

Insert the path *loc* at the start of the list of directories in which to search for Fortran 90 `INCLUDE` files.

- No space is allowed between `-I` and *loc*.
- Invalid directories are just ignored with no warning message.
- The `-Iloc` applies to `INCLUDE` files with *relative*, not *absolute*, path names.

Example: `f90 -I/usr/applib growth.f90`

Above, `f90` searches for `INCLUDE` files in the source file directory and then in the `/usr/applib` directory.

Use `-Iloc` again to insert more paths.

Example: `f90 -Ipath1 -Ipath2 any.f90`

Search Order: The search for `INCLUDE` files is in the following order:

- The directory containing the source file
- Directories named in `-Iloc` options

`-l x` Library—link with library `libx`.

Pass `-l x` to the linker. `ld` links with object library `libx`. If shared library `libx.so` is available, `ld` uses it, otherwise, `ld` uses archive library `libx.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 `libabc`.

```
demo$ f90 any.f90 -labc
```

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

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

```
demo$ f90 any.f90 -ly -lz
```

See also Section 6.5, “Library Paths,” on page 65.

-Ldir Library—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 “Building Executables: `ld` Search order” on page 66. For the relative order between `LD_LIBRARY_PATH` and `-Ldir`, see `ld(1)`.

Example: Use `-Ldir` to specify a library search directory.

```
demo$ f90 -Ldir1 any.f90
```

Example: Use `-Ldir` again to add more directories.

```
demo$ f90 -Ldir1 -Ldir2 any.f90
```

Restrictions

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

-mt Multithread safe libraries—use for low level threads.

Use multithread safe libraries. If you do your own low level thread management this helps prevent conflicts between threads. Use `-mt` if you mix C and Fortran 90 and you do your own thread management of multithread C coding using the `libthread` primitives. Before you use your own multithreaded coding, read the “*Guide to Multi-Thread Programming*.”

The `-mt` option does not require the Multiprocessor Fortran 90 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.

The Fortran 90 library `libf90` is multithread safe. The FORTRAN 77 library that is linked in if you use `-mt`, `libF77_mt`, is also multithread safe.

Restrictions for -mt

- 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.

`-mdir` Modules—look for Fortran 90 modules in the *dir* directory also.

No space is allowed between `-M` and *dir*.

By default, such files are sought in the current working directory. The `-mdir` option allows you to keep them in some other location in addition.

Background—If a file containing a Fortran 90 module is compiled, `f90` generates a module file (`.M` file) in addition to the `.o` file.

`-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`.

-nolib No automatic libraries.

Do *not* automatically link with *any* system or language library; that is do *not* pass any `-lx` 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 `libm` statically and link `libc` dynamically.

```
demo$ f90 -nolib any.f90 -lf90 -Bstatic -lm -Bdynamic -lc
```

There is no dynamic `libf90`; it is always linked statically.

Order for `-lx` options is important. Use the order shown in the example.

-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.

-norunpath No run path in executable.

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 `-Rlist`.

-o nm Output file—rename the output file.

Name the final output file *nm* instead of *a.out*. There must be a blank between *-o* and *nm*.

-onetrip DO loops—use one trip DO loops.

Compile DO loops so that they are performed at least once if reached. DO loops in Fortran 90 ordinarily are not performed at all if the upper limit is less than the lower limit, unlike *some* implementations of FORTRAN 66 DO loops.

-O[n] Optimize the object code for speed of execution.

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 suppresses *-On*.

-O If you do not specify an *n*, *f90* uses whatever *n* is most likely to yield the fastest performance for most reasonable applications. For the current release, this is 3.

-O1 Do only conservative scalar optimization.

This level usually results in moderate code size and compile time. If an optimization can create a false exception, then that optimization is not performed. At this level, *f90* can analyze whether a variable is used before it is defined.

-O2 Do moderate optimization.

-O3 Do aggressive scalar optimization.

Some of these optimizations can create false exceptions. This level can result in larger code size and compile time. At this level, *f90* can analyze whether a variable is used before it is defined.

-O4 For this release, -O4 is equivalent to -O3.

-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` is provided mostly for compatibility with older systems. `-pg` with `gprof` does more.

-parallel Multiprocessor—Parallelize automatically and parallelize explicitly indicated loops.

Parallelize loops both automatically by the compiler and as explicitly specified by the programmer. With explicit parallelization of loops, there is a risk of producing incorrect results.

If optimization is not at -O3, then it is raised to -O3.

Restrictions:

- `-g` turns off `-parallel`.
- Avoid `-parallel` if you do your own thread management. See `-mt`.
- Do not mix parallelized `f77` and parallelized `f90`.
- If you use `-parallel` and compile and link in *separate* steps, then *link* with `-parallel`.

The `-parallel` option requires the Multiprocessor Fortran 90 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.

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 B.4, “Number of Processors.”

Before you use `-parallel`, see Appendix B, “iMPact: Multiple Processors,” Appendix C, “iMPact: Automatic Parallelization,” and Appendix D, “iMPact: Explicit Parallelization.”

-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`.

-Qoption pr op Option—pass option list to specified program.

Pass the option list `op` to the program `pr`. There must be a blank between `Qoption` and `pr` and `op`. 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 assembler used by the compiler is named `fbe`.

Example: Pass the help option for help to the linker, `ld`.

```
demo$ f90 -Qoption ld -Dhelp src.f90
```

Example: Pass the load map to the linker, `ld`.

```
demo$ f90 -Qoption ld -m src.f90
```

-reduction Multiprocessor—reduction loops: analyze loops for reduction.

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

The `-reduction` option requires the Multiprocessor Fortran 90 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 `-reduction`, see Appendix C, “iMPact: Multiple Processors” and Appendix D, “iMPact: Automatic Parallelization.”

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

Example: Automatically parallelize with *reduction*.

```
demo$ f90 -parallel -reduction any.f90
```

-R *ls* Library paths—store paths into object file.

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

- *ls* is a colon-separated list of directories for library search paths.
- The blank between `-R` and *ls* 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 `f90`, `-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 90 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).

-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.

Parallel CALL: This option gives more freedom to the optimizer for such tasks as parallelizing a loop that includes a `CALL`.

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

- Arguments in a `SUBROUTINE` or `FUNCTION` statement (already on stack)
- 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/`

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 MBytes for the main stack and 256 KBytes 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.

The main stack size →

Example: *Stack size*—show the current *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 MBytes.

```
demo% limit stacksize 65536
```

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

```
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 `f90` 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.

-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.

-V Version ID—show version ID.

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

-w[*n*] Warnings—suppress warnings.

The *n* can be any one of 0, 1, 2, 3, or 4.

- -w0 suppresses the least warnings.
- -w4 suppresses the most warnings.
- -w with no *n*, is the same as -w0.

-xlibmopt 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$ f90 -c -xlicinfo any.f90
```

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

```
demo$ f90 -xlicinfo -c any.f90
```

-xnolib Synonym for -nolib.

-xnolibmopt 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:
f90 -fast -xnolibmopt ...

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

-xpg Synonym for -pg.

-xtime Synonym for -time.

-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 necessarily prevent the program from being compiled and linked.

Table 3-4 Summary of -Xlist Suboptions

Option	Action
-Xlist <i>(no suboption)</i>	Errors, listing, and cross reference table.
-XlistE	Errors.
-Xlisterr	Suppress all error messages in the verification report.
-Xlisterr[nnn]	Suppress error <i>nnn</i> in the verification report.
-XlistI	Include files.
-XlistL	Listing (and errors).
-Xlistln	Page length is <i>n</i> lines.
-Xlisto <i>name</i>	Rename the -Xlist output report file.
-Xlistwar	Suppress all warning messages in the report.
-Xlistwar[nnn]	Suppress warning <i>nnn</i> in the report.
-XlistX	Cross reference table (and errors).

For more information, see “Details of -Xlist Suboptions” on page 75.

3.3 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 `f90f`.

```
demo$ alias f90f "f90 -fast -O4"
```

Example: Use `f90f`.

```
demo$ f90f any.f90
```

The above command is equivalent to “`f90 -fast -O4 any.f90`”.

Environment Variable Method

Some users shorten command lines by using environment variables. The `FFLAGS` or `OPTIONS` variables are 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 '-fast -O4'
```

- Example: Use FFLAGS explicitly.

```
demo$ f90 $FFLAGS any.f90
```

The above command is equivalent to “f90 -fast -O4 any.f90”.

- Example: Let make use FFLAGS implicitly.

If both:

- The compile in a make file is *implicit* (no *explicit* f90 compile line)
- The FFLAGS variable is set as above

Then invoking the make file results in a compile command equivalent to “f90 -fast -O4 any.f90”.

File System and File I/O



This chapter is organized into the following sections.

<i>Summary</i>	<i>page 35</i>
<i>Directories</i>	<i>page 37</i>
<i>File Names</i>	<i>page 37</i>
<i>Path Names</i>	<i>page 37</i>
<i>Redirection</i>	<i>page 40</i>
<i>Piping</i>	<i>page 41</i>
<i>Accessing Files from Fortran Programs</i>	<i>page 45</i>
<i>Direct I/O</i>	<i>page 45</i>
<i>Internal Files</i>	<i>page 46</i>

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.

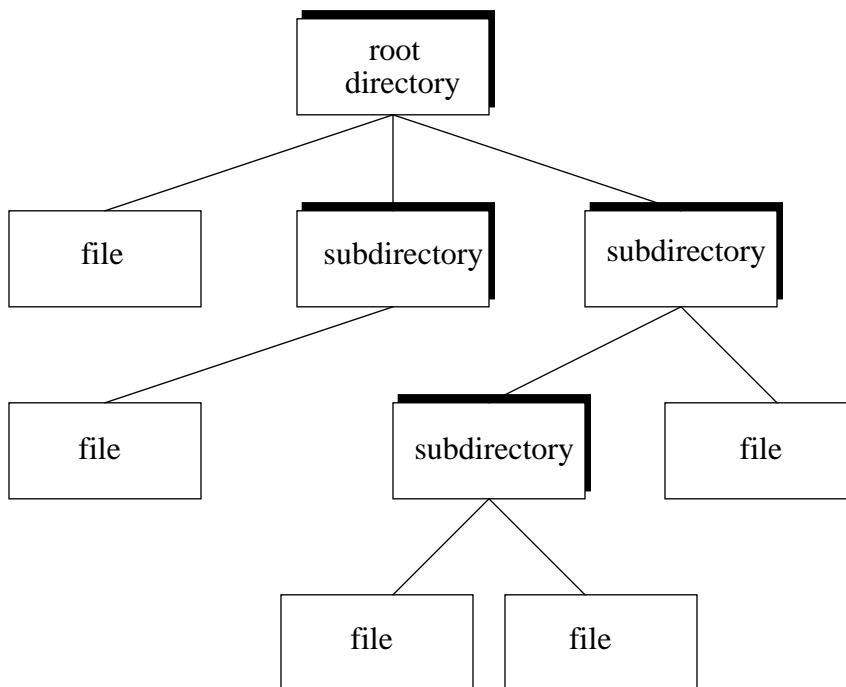


Figure 4-1 File System Hierarchy

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:

`/usr/you`

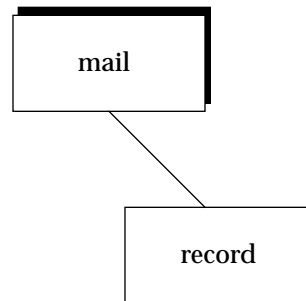


Figure 4-2 Relative Path Name

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`

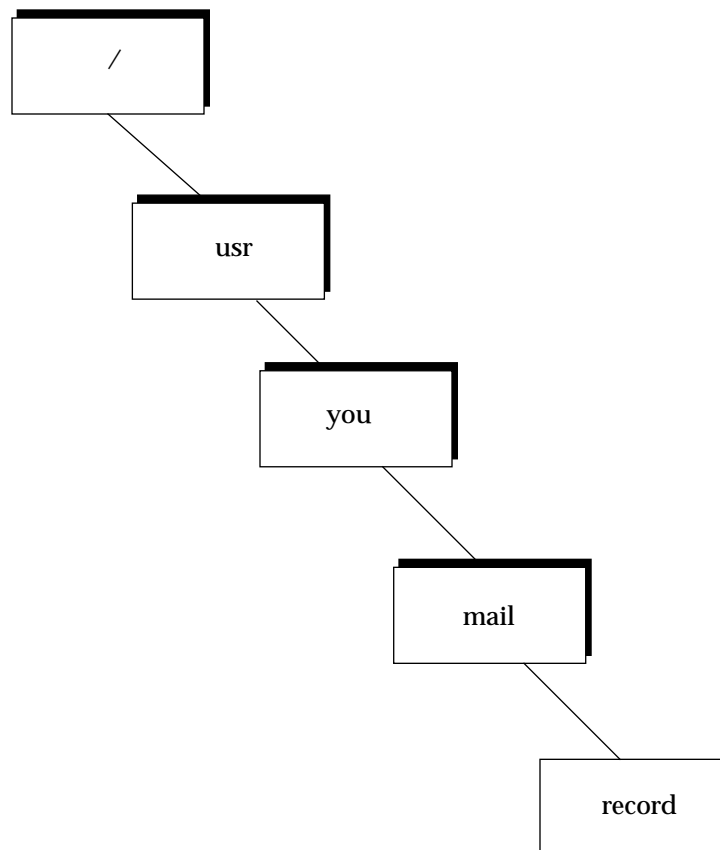


Figure 4-3 Absolute Path Name

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 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:

```
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 90 program and reads from unit 5, it reads from the `mydata` file.

Output/Truncate

The shell command line:

```
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 90 program `myprog` writes to unit 6, it writes to the file `myoutput`.

Output/Append

The shell command line:

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

The above command causes the file `myoutput` (which must exist) to be connected for *appending*. So if the Fortran 90 program `myprog` writes to unit 6, it writes to the file `myoutput` but after wherever the file ended before.

Both standard input and standard output may be redirected to and from different files on the same command line. 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 standard error and standard output.

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

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

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

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

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*.

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`.

4.7 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 90 OPEN statement establishes a connection between the program and file to be accessed. (For a description of OPEN, read the *Fortran 90 Handbook*.)

File names can be simple expressions, as listed below.

- Quoted character constants

```
File = 'myfile.out'
```

- Character variables

```
File = Filnam
```

- Character expressions

```
File = LEN_TRIM(Prefix) // '/' // LEN_TRIM(Name)
```

A program can read file names from a file or terminal keyboard.

```
READ( 4, 401 ) Filnam
```


Accessing Unnamed Files

When a program opens a Fortran 90 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.FAAAxnnnnn`, 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 90 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 `OPENS` 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 there was no way to open files by name.

With this operating system the following ways communicate file names to a Fortran program.

- Redirection and piping. Redirection and piping can change the names of program input and output files without changing the program. See the sections “Redirection” and “Piping” earlier in this chapter.

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, the following are preconnected.

- 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.f90`.

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

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

```
demo$ f90 OtherUnit.f90
demo$ a.out
demo$ cat fort.25
      2
demo$
```

4.8 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: Open, 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)”.

4.9 *Internal Files*

An internal file is a variable of type default character. This means that an internal file can be one of the following:

- Scalar
- Array
- Element of an array
- Section of an array
- Component of a structure
- Substring

To use an internal file, give the name or the designator of the character variable in place of the unit number.

This is called I/O, because you use `READ` and `WRITE` statements to deal with internal files, although I/O is not a precise term to use here.

£90 extends what can be an internal file: if you are *reading* from an internal file, the internal file can be a *literal constant* character string.

Rules and Restrictions

- The variable must not be an array section with a vector subscript.
 - For a constant there is a single record in the file. ♦
 - For a variable or substring, there is a single record in the file.
 - For an array, each array element is a record.
 - Each sequential `READ` or `WRITE` starts at the beginning of an internal file.

Example: Scalar for internal file, sequential formatted read.

```
demo$ cat intern1.f90
CHARACTER x*80
WRITE(*,*) 'Enter two numbers'
READ( *, '(A)' ) x      ! Reads a character string from standard input to x
READ( x, '(I3,I4)' ) n1, n2 ! Reads the internal file x
WRITE( *, * ) n1, n2
END
demo$ f90 intern1
demo$ a.out
Enter two numbers
12 99
12 99
demo$ █
```

Example: Array for internal file, sequential formatted read.

```
demo$ cat intern3.f90
CHARACTER *16, Line(4)
DATA Line / ' 81 81 ', ' 82 82 ', ' 83 83 ', ' 84 84 ' /
READ( Line, '(2I4)' ) i, j, k, l, m, n ! Reads internal file Line
PRINT *, i, j, k, l, m, n
END
demo$ f90 intern3
demo$ a.out
81 81 82 82 83 83
demo$ █
```


Program Development



This chapter is organized into the following sections.

<i>Simple Program Builds</i>	<i>page 49</i>
<i>Program Builds with the make Program</i>	<i>page 50</i>
<i>Tracking and Controlling Changes with SCCS</i>	<i>page 52</i>

5.1 Simple Program Builds

For a program that depends on a few source files and some system libraries, you can easily compile all of the source files every time you change the program. Even in this simple case, the `f90` command can involve much typing, and with options or libraries, a lot to remember. A script or alias can help.

Writing a Script

A shell script can save typing. For example, to compile a small program that is in the file `example.f90`, and that uses the Xview support library, you can save a one-line shell script onto a file, here called `fex`, that looks like this.

```
f90 example.f90 -lFxview -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 "f90 example.f90 -lFxview -o example"
```

Using a Script or Alias

Either way, to recompile `example.f90`, 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.

5.2 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 `make` File

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 `make` file, in the directory where you are developing the program.

Example: A program of four source files and a `make` file.

```
demo$ ls
makefile
commonblock
computepts.f90
pattern.f90
startupcore.f90
demo$ █
```


Assume both `pattern.f90` and `computepts.f90` do an include of `commonblock`, and you wish to compile each `.f90` file and link the three relocatable files (plus a series of libraries) into a program called `pattern`.

The make file for this example is listed below.

```
demo$ cat makefile
pattern: pattern.o computepts.o startupcore.o
        f90 pattern.o computepts.o startupcore.o -Fxview -o pattern
pattern.o: pattern.f90 commonblock
        f90 -c pattern.f90
computepts.o: computepts.f90 commonblock
        f90 -c computepts.f90
startupcore.o: startupcore.f90
        f90 -c startupcore.f90
demo$ █
```

The first line of this make file 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 (because it starts with a tab).

There are four such paragraphs or entries in this make file. 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.

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 `make` file, 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 `make` file for that target.

5.3 *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`.

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 `make` files, the source is all there is, so the SCCS information can go in comments, if anywhere.

Identify the `make` file with a `make` comment containing the keywords.

```
#      %Z%M% %I% %E%
```

The source files `startupcore.f90` and `computepts.f90` and `pattern.f90` can be identified to SCCS by initialized data of the form.

```
CHARACTER (LEN=50) :: sccsid = "%Z%M% %I% %E%\n"
```

Creating SCCS Files

Example: Put files under control of SCCS with an SCCS `create` command.

```
demo$ sccs create makefile commonblock startupcore.f90 \  
      computepts.f90 pattern.f90  
demo$ █
```

The make file looks like this after SCCS keyword expansion.

```
#    @(#)makefile1.184/03/01
OBJ = pattern.o computepts.o startupcore.o
pattern: $(OBJ)
        f90 $(OBJ) -Fxview -o pattern
pattern.o: pattern.f90 commonblock
computepts.o: computepts.f90 commonblock
startupcore.o: startupcore.f90
```

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.f90
demo$ █
```

In this example, SCCS makes a writable copy of `computepts.f90` 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 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.f90
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.f90` 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.

Libraries



This chapter is organized into the following sections.

<i>Libraries in General</i>	<i>page 57</i>
<i>Static Libraries</i>	<i>page 58</i>
<i>Dynamic Libraries</i>	<i>page 61</i>
<i>Consistent Compile and Link</i>	<i>page 65</i>
<i>Library Paths</i>	<i>page 65</i>

See `ld(1)` for more details.

6.1 Libraries in General

A *library* can be a collection of subprograms. Each member of this collection 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.

Some examples of *static* libraries on the system are:

- Fortran 90 library: `libf90.a`
- Math library: `libm.a`
- C library: `libc.a`

Some examples of *shared dynamic* libraries on the system are:

- FORTRAN 77 library: `libF77.so`
- C library: `libc.so`

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. This is a very simple load map.

Example: `-m` for load map.

```
demo$ f90 -Qoption ld -m any.f90
```

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 (at least, for *static* libraries).
- The programmer need not change the link command line as subroutine calls are added and removed during program development.

6.2 Static Libraries

Static libraries are built from object files (`.o` files) using the program `ar`.

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 nature of the elements and the nature of the search leads to some disadvantages.

- *The whole thing* — For *static* libraries, when the linker extracts a library element, it extracts the whole thing (not so for *dynamic* libraries). A library element corresponds to the result of a compilation, so routines that are compiled together are always linked together. This is a difference between this operating system and some other systems, and it may affect whether you chunk compilation files to many small files for your libraries.
- *Order matters* — In linking *static* libraries, order really matters. The linker processes its input files in the order that they appear on the command line (that is, 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.

You can use `lorder` and `tsort` to order static libraries.

Example: *Order matters*. If a Fortran 90 program is in two files, `main.f90` and `graf.f90`, and only the latter accesses the Xview library, it is an error to reference that library before `graf.f90` or `graf.o`:

```
(Wrong) demo$ f90 main.f -lXview graf.f90 -o myprog
(Right) demo$ f90 main.f graf.f90 -lXview -o myprog
```

Sample Creation of a Static Library

Base—source of four routines (for example on creating static library).

```
demo$ cat one.f
      SUBROUTINE twice ( a, r )
      REAL a, r
      r = a * 2.0
      END
      SUBROUTINE half ( a, r )
      REAL a, r
      r = a / 2.0
      END
      SUBROUTINE thrice ( a, r )
      REAL a, r
      r = a * 3.0
      END
      SUBROUTINE third ( a, r )
      REAL a, r
      r = a / 3.0
      END
demo$ █
```

Main program to use one of the routines (for creating static library).

```
demo$ cat teslib.f
      READ(*,*) x
      CALL twice( x, z )
      WRITE(*,*) z
      END
demo$ █
```

- Split the file, using `fsplit`, so there is one subroutine per file.

```
demo$ fsplit one.f    ← fsplit assumes fixed form. It may not work on all f90 source.
demo$ █
```

- Compile each with `-c` so it will *compile only*, and leave the `.o` object files.

```
demo$ f90 -c half.f
demo$ f90 -c third.f
demo$ f90 -c thrice.f
demo$ f90 -c twice.f
demo$ █
```

- Use `ar` to create static library `faclib.a` from the four object files.

```
demo$ ar cr faclib.a half.o third.o thrice.o twice.o
```

Alternate: Specify the order using `lorder` and `tsort`.

```
demo$ ar cr faclib.a 'lorder half.third.o thrice.o twice.o | tsort'
```

- Compile the main, using the new static library.

```
demo$ f90 teslib.f90 faclib.a
demo$ █
```

- Use `nm` to list the name of each object in `a.out` built from static library.

```
demo$ 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$ █
```

Create the library.

Note that `twice` is here.

Note that
`half`, `third`, and `thrice`
are not here (good).

- 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've changed the source), replace it in its library by running `ar` again. A library need not be specially flagged for the linker; the linker recognizes a library when it encounters one.

Example: Recompile, replace. Give `ar` the `r` option; use `cr` only for creating.

```
demo$ f90 -c half.f
demo$ ar r faclib.a half.o
demo$ █
```

6.3 *Dynamic Libraries*

A dynamic library has the following features.

- It is a collection of object modules such that each is already in executable file format (the `a.out` format) but the collection 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.
- If you change a module of a shared library, then whenever any application using it starts to execute, it will get the changed version. The ability to modify and improve libraries independent of the executables that use it can be a significant advantage in maintaining programs.

If you have a `-pic` option to use to generate position-independent code, then from the generated dynamic library, a module can be used by many executing programs *without* duplicating it in them all. In this release, `f90` has no `-pic` to generate position-independent code, so the library is not truly shared. You still save disk space for storing the executable, and you can still get library changes into the executable without rebinding the executable.

Performance Issues

The usual trade-off between space and time applies.

- Less space

In general, deferring the binding of the library module uses less *disk* space

- 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.

- Programs Vary

Because of these various performance issues, some programs are slower if they use nonshared libraries, and some if they use shared libraries. You might bind each way to tell whether one method is significantly better for your program.

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.

-d[y,n]: Allow or disallow dynamic libraries for the entire executable

- -dy: Yes—allow *dynamically* bound libraries (*allow* shared libraries).
- -dn: No—do *not* allow dynamically bound libraries (*no* shared libraries).

These apply to the *whole* executable. Use only *once* on the command line. The default is *y*.

-B[dynamic,static]: Bind as dynamic (or static) libraries listed later

This applies to any library listed later in the command. The default is *dynamic*.

- -Bdynamic: Prefer *dynamic* binding (try for shared libraries).
- -Bstatic: Require *static* binding (no shared libraries).

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.

A Simple Dynamic Shared Library

You can build a shared library from the relocatable object (`.o`) files using the `ld` command. Be careful to avoid any need for reentrant code, since this release does not provide a way to guarantee that code is position independent.

Sample Create

Example: Create a dynamic shared library.

Start with the same files used for the static library example: `half.f90`, `third.f90`, `thrice.f90`, `twice.f90`.

Compile.

```
demo$ f90 -c *.f90
demo$ █
```

Example: Create a dynamic shared library.

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 *.o
demo$ █
```

The `-G` tells the linker to build a shared library.

Note that there is no “`-z text`” because that would generate warnings. 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.

Bind: Make the executable file `a.out`.

```
demo$ f90 teslib.f90 libfac.so.1
demo$ █
```

Run.

```
demo$ a.out
6
      12.0000
demo$ █
```

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.

```
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 `f90`). It also shows exactly which files on the system will be used for these libraries.

```
demo$ ldd a.out
libfac.so.1 => ./libfac.so.1
libF77.so.2 => /opt/SUNWsprow/lib/libF90.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.

6.4 Consistent Compile and Link

Be consistent with compiling and linking. Do not build libraries with inconsistent modules. Inconsistent compilation and linkage is not supported.

- For compiling and linking as separate steps (separate commands), if you *compile* any module with `-g`, then be sure to *link* with the same option.

Example. Compile `sbr.f90` with `-g` and `smain.f90` without it.

```
demo$ f90 -c -g sbr.f90
demo$ f90 -g sbr.o smain.f90    ← Pass the -g to the linker.
```

The above sequence is equivalent to the following commands.

```
demo$ f90 -c -g sbr.f90
demo$ f90 -c smain.f90
demo$ f90 -g sbr.o smain.o
```

- If you compile *any* module under a major release of the operating system, then compile *all* modules of that program with the same major release.

6.5 Library Paths

The linker searches for libraries in several locations and it searches in certain prescribed orders. You can make some changes to the order and locations.

Installation Directory

Some library locations depend on the installation directory. These locations are described here in terms of a path called *BasDir*, defined as follows:

Installation Location	<i>BasDir</i>
Standard	<code>/opt/SUNWspro/lib/</code>
Nonstandard, for example, <code>/my/dir/</code>	<code>/my/dir/</code>

Building Executables: ld Search order

During the *build* of the executable, ld searches for libraries in the following locations, in order:

Path	Comment
/BasDir/lib/	Sun shared libraries here
/BasDir/SC3.0.1/lib/	Sun libraries, shared or static, here
/opt/SUNWspro/lib/	Standard location for Sun libraries
/usr/ccs/lib/	
/usr/lib/	

The linker also searches in any directories specified in LD_LIBRARY_PATH or by the -Ldir option.

Running Executables: ld Search order

During the *run* of the executable, ld searches for shared libraries in the following locations, in order:

Path	Comment
/BasDir/lib/	Built in by driver, unless -norunpath
/opt/SUNWspro/lib	Built in by driver, unless -norunpath
Directories built in by -R or LD_RUN_PATH when executable was made	
/usr/ccs/lib/	
/usr/lib/	

The linker also searches in directories specified in LD_LIBRARY_PATH.

Summary of Environment Variables for Paths

- LD_RUN_PATH: Value matters only when executable is *created*.
- LD_LIBRARY_PATH: Value matters when executable is *created* or *run*.

Build Paths and Run Paths

When you run the executable file, the runtime linker locates the shared libraries again. The linker searches in any directories specified in the `LD_LIBRARY_PATH` environment variable, and that variable can change even after the executable file has been created. Therefore it doesn't matter where the libraries were when the executable was created.

Finding Built-in Paths

Use `dump` to check which paths were built in when the executable was created.

Example: Find which directories were built in to `a.out`.

```
demo$dump -Lv a.out | grep RPATH
```


Debugging

This chapter is organized into the following sections.

<i>Global Program Checking (-Xlist)</i>	<i>page 69</i>
<i>The dbx Debugger</i>	<i>page 77</i>

7.1 Global Program Checking (-Xlist)

Purpose—Checking across routines helps find various kinds of bugs.

With `-Xlist`, `f90` 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
Form of output varies.

```
demo$ f90 -XlistE Repeat.f90
demo$ cat Repeat.lst
FILE  "Repeat.f90"
...
demo$ █
```

Errors in General

Global program checking can do the following:

- Enforce type checking rules of Fortran 90 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 90 program
- Portability problems—codes that do not conform to ANSI Fortran 90, 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$ f90 -Xlist any1.f90 any2.f90 any3.f90
```

In the above example, `f90` 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 90 files for global program checking.

```
demo$ f90 -Xlist *.f90
```

Terminal Output

To display directly to the terminal, rename the output file to `/dev/tty`.

Example: Display to terminal.

```
demo$ f90 -Xlisto /dev/tty any1.f90
```

See *-Xlisto name*, on page 76.

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 `f90` 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)

Example: Using `-xlist`—a program with inconsistencies between routines.

Repeat.f90

Form of output varies.

```
demo$ cat Repeat.f90
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$ f90 -xlist Repeat.f90
demo$ cat Repeat.lst
```

Compile with `-xlist`. →
List the `-xlist` output file. →

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: `-XlistE`, `-XlistL`, and `-XlistX`.
- All other options are detail options, not *a la carte*, 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$ f90 -XlistL -Xlist any.f90
demo$ f90 -XlistL any.f90
```

Combination special or *a la carte* suboptions (with no other suboptions):

Table 7-1 -Xlist Combination Special or A La Carte Suboptions

Type/Amount of Output	Option	Comment	Details
Errors, listing, cross reference table	-Xlist	No suboptions	page 71
Errors	-XlistE	By itself, does not show listing or cross reference table	page 75
Errors and listing	-XlistL	By itself, does not show cross reference table	page 76
Errors and cross reference table	-XlistX	By itself, does not show listing	page 76

Summary of -Xlist Suboptions

Table 7-2 -Xlist Suboptions Summary

Option	Action	Details
-Xlist (no suboption)	Errors, listing, and cross reference table.	page 73
-XlistE	Errors.	page 75
-Xlisterr	Suppress all error messages in the verification report.	page 75
-Xlisterr[nnn]	Suppress error <i>nnn</i> in the verification report.	page 75
-XlistI	Include files.	page 75
-XlistL	Listing (and errors).	page 76
-Xlistln	Page length is in <i>n</i> lines.	page 76
-Xlisto name	Rename the -Xlist output report file.	page 76
-Xlistwar	Suppress all warning messages in the report.	page 76
-Xlistwar[nnn]	Suppress warning <i>nnn</i> in the report.	page 76
-XlistX	Cross reference table (and errors).	page 76

Details of -Xlist Suboptions

- `-Xlisterr` Suppress *all* error messages in the verification report.
- `-Xlisterr[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 not 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.
- `-XlistE` Global cross check errors. Show cross routine errors. This suboption by itself does not show a listing or a cross reference.
- `-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
 - Cross Reference Table

If the cross reference table is not suppressed, then the following are all scanned while generating the cross reference table:

 - Source files
 - INCLUDE files
- Default: No include files.

-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).

-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.

-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**

-Xlistwar Suppress *all* warning messages in the report.

-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

Suppress specific warnings.

```
demo$ f90 -Xlistwar338 -Xlistwar348 -XlistE Repeat.f90
demo$ cat Repeat.lst
FILE "Repeat.f90"
demo$ █
```

7.2 The dbx Debugger

This section is organized as follows:

<i>Sample Program for Debugging</i>	<i>(example)</i>	<i>page 78</i>
<i>A Sample dbx Session</i>	<i>(example)</i>	<i>page 79</i>
<i>Segmentation Fault—Finding the Line Number</i>	<i>(example)</i>	<i>page 81</i>
<i>Exception—Finding the Line Number</i>	<i>(example)</i>	<i>page 83</i>
<i>Trace of Calls</i>	<i>(example)</i>	<i>page 84</i>
<i>Pointer to a Scalar</i>	<i>(example)</i>	<i>page 85</i>
<i>Pointer to an Array</i>	<i>(example)</i>	<i>page 86</i>
<i>User-Defined Types</i>	<i>(example)</i>	<i>page 87</i>
<i>Pointer to User-Defined Type</i>	<i>(example)</i>	<i>page 89</i>
<i>Allocated Arrays</i>	<i>(example)</i>	<i>page 91</i>
<i>Print Arrays</i>	<i>(example)</i>	<i>page 92</i>
<i>Print Array Slices</i>	<i>(example)</i>	<i>page 93</i>
<i>Generic Functions</i>	<i>(example)</i>	<i>page 94</i>
<i>Miscellaneous Tips</i>		<i>page 96</i>
<i>Main Features of the Debugger</i>		<i>page 96</i>
<i>Help</i>	<i>(example)</i>	<i>page 97</i>

This section *introduces* some dbx features likely to be used with Fortran. Use it as a *quick start* for debugging Fortran. Be sure to try the `help` feature.

Note – Before you use the Debugger, you must install the appropriate Tools package—read *Installing SunSoft Developer Products on Solaris* for details.

With dbx you can display and modify variables, set breakpoints, trace calls, and invoke procedures in the program being debugged without having to recompile.

The Debugger program lets you make more effective use of dbx by replacing the original, terminal-oriented interface with a window- and mouse-based interface.

Sample Program for Debugging

The following program, with bug, and consisting of files a1.f90, a2.f90, and a3.f90, is used in several examples of debugging.

Example: Main for debugging.

```
a1.f90  PROGRAM TryDbx
        INTEGER, PARAMETER :: n=2
        REAL, DIMENSION(n,n) :: twobytwo
        DATA (( twobytwo(k,j),k=1,n),j=1,n) / 4*-1 /
        CALL mkidentity( twobytwo, n )
        PRINT *, determinant( twobytwo )
END
```

Example: Function for debugging.

```
a3.f90  REAL FUNCTION determinant ( a )
        REAL a(2,2)
        determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
END
```

Example: Subroutine for debugging.

```
a2.f90  SUBROUTINE mkidentity ( array, m )
        REAL, DIMENSION(m,m) :: array
        DO i = 1, m
            DO j = 1, m
                IF ( i .eq. j ) THEN
                    array(i,j) = 1.0
                ELSE
                    array(i,j) = 0.0
                END IF
            END DO
        END DO
END
```

A 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$ f90 -o silly -g a1.f90 a2.f90 a3.f90
```

Example: Compile and link *in separate steps*.

```
demo$ f90 -c -g a1.f90 a2.f90 a3.f90
demo$ f90 -o silly a1.o a2.o a3.o
```

- **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, at the dbx prompt; type “stop in *subnam*”, where *subnam* names a subroutine or function, and *subnam* can be upper case or lower case.

Example: A way to stop at the first executable statement in a main program.

```
(dbx) stop in main
(2) stop in main
(dbx) █
```

- **Run Program**—To run a program from within dbx, enter the `run` command.

Example: Run a program from within dbx.

```
(dbx) run
Running: silly
(process id 8786)
stopped in main at line 5 in file "a1.f90"
      5      CALL mkidentity( twobytwo, n )
(dbx) ■
```

When the breakpoint is reached, dbx displays a message showing where it stopped, in this case at line 5 of the `a1.f90` file.

- **Print**—To print a value, enter the `print` command.

Example: Print the variable `n`. Note that dbx handles parameters.

```
(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.

```
(dbx) print array
dbx: "array" is not defined in the current scope
dbx: see 'help scope' for details
(dbx) ■
```

In the above example:

- The print fails because `array` is not defined here—only in `mkidentity`.
- The error message details may vary with the release, and translation.

- *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 6 in file "a1.f90"
      6      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$ ■
```

Note that `print twobytwo` now displays the unit matrix.

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

The most frequent causes are the following:

- An array index being outside the declared range
- 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

You can locate the offending source line using `-Xlist` or `dbx`.

- Recompile with the `-Xlist` option to get global program checking
- Use `dbx` to find the source code line where a segmentation fault occurred

Example: Program to generate a segmentation fault.

```
demo 4% cat WhereSEGV.f90
      INTEGER a(5)
      j = 2000000
      DO i = 1,5
         a(j) = (i * 10)
      END DO
      PRINT *, a
      END
demo 5% █
```

Example: Use `dbx` to locate a segmentation fault.

```
demo 5% f90 -g WhereSEGV.f90
demo 6% a.out
Segmentation fault (core dumped)
demo 7% dbx a.out
Reading symbolic information for a.out
... other messages ...
(dbx)run
Running: a.out
(process id 8813)
signal SEGV (no mapping at the fault address) in main at line 4
      in file "WhereSEGV.f90"
      4              a(j) = (i * 10)
(dbx) quit
demo 8% █
```


Exception—Finding the Line Number

Example: Find where an exception occurred.

WhereExcept.f90

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 ieeeer, ieee_handler, myhandler
REAL :: r=14.2, s=0.0
ieeeer = ieee_handler('set', 'all', myhandler)
PRINT *, r/s
END
INTEGER FUNCTION myhandler(sig, code, context) ! Handler
!
  INTEGER sig, code, context(5)
  CALL abort()
END
demo$ f90 -g WhereExcept.f90
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) █

```

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.f90 is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

Execution stopped, line 23 →

calcb called from calc, line 9 →
calc called from main, line 3 →
Note reverse order: main called calc, calc called calcb.

```
demo$ f90 -g ShowTrace.f90
demo$ a.out
Segmentation Fault (core dumped)
demo$ dbx a.out
Reading symbolic information for a.out
...
(dbx) run
(process id 8939)
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.f90"
    [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f90"
    [3] main(), line 3 in "ShowTrace.f90"
(dbx) █
```

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.

Pointer to a Scalar

Example: Pointer to a scalar, in dbx.

```

PtrScal.f90
demo% f90 -g PtrScal.f90
demo% dbx a.out
(dbx) list 1,99
1  PROGRAM PtrScalar
2  REAL, POINTER :: p
3  REAL, TARGET :: r
4  p => r
5  r = 2.3
6  PRINT *, p
7  p = 3.2
8  PRINT *, r
9  END
(dbx) stop at 8
(2) stop at "PtrScal.f90":8
(dbx) run
Running: a.out
(process id 12367)
2.29999995
stopped in main at line 8 in file "PtrScal.f90"
8  PRINT *, r
(dbx) whatis p
real*4 p ! f90 pointer
(dbx) whatis r
real*4 r
(dbx) print p
p = 3.2
(dbx) print r
r = 3.2
(dbx) quit
demo$
```

Pointer to an Array

Example: Pointer to an array, in dbx.

```
PtrArray.f90
demo% f90 -g PtrArray.f90
demo% dbx a.out
(dbx) list 1,99
1  PROGRAM PtrArray
2      INTEGER, TARGET :: a(5,5)
3      INTEGER, POINTER :: corners(:, :)
4      DO i = 1,5
5          a(i,:) = i
6      END DO
7      corners => a(1:5:4, 1:5:4)
8      PRINT *, corners
9  END
(dbx) stop at 8
(2) stop at "PtrArray.f90":8
(dbx) run
Running: a.out
(process id 12397)
stopped in main at line 8 in file "PtrArray.f90"
8      PRINT *, corners
(dbx) whatis a
integer*4 a(1:5,1:5)
(dbx) whatis corners
integer*4 , corners(1:2,1:2) ! f90 pointer
(dbx) print corners
corners =
(1,1)      1
(2,1)      5
(1,2)      1
(2,2)      5
(dbx) quit
demo$
```

User-Defined Types

Example: Structures—user defined types, in dbx.

```

demo% f90 -g DebStruc.f90
demo% dbx debstr
DebStruc.f90 (dbx) list 1,99
1  PROGRAM Struct  ! Debug a Structure
2      TYPE product
3          INTEGER      id
4          CHARACTER*16  name
5          CHARACTER*8   model
6          REAL          cost
7          REAL          price
8      END TYPE product
9
10     TYPE(product) :: prod1
11
12     prod1%id = 82
13     prod1%name = "Schlepper"
14     prod1%model = "XL"
15     prod1%cost = 24.0
16     prod1%price = 104.0
17     WRITE ( *, * ) prod1%name
18     END
(dbx) stop at 17
(2) stop at "Struct.f90":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f90"
17     WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
integer*4 id
character*16 name
character*8 model
real*4 cost
real*4 price
end type product
(dbx)

```

Example: Structures—user-defined types, in dbx.

```
(dbx) print prod1
prod1 = (
    id      = 82
    name    = 'Schlepper'
    model   = 'XL'
    cost    = 24.0
    price   = 104.0
)
(dbx) quit
(dbx) ■
```

Pointer to User-Defined Type

Example: Structures—user defined types, and pointers, in dbx.

DebStruc.f90
 Declare a user-defined type.
 Declare variables *prod1* and *prod2* to be of that type and targets.
 Declare variables *curr* and *prior* as pointers to the type.
 Make *curr* point to *prod1*.
 Make *prior* point to *prod1*.
 Initialize *prior*.
 Set *curr* to *prior*.
 Print *name* from *curr* and *prior*.

```

demo% f90 -o debstr -g DebStruc.f90
demo% dbx debstr
(dbx) stop in main
(2) stop in main
(dbx) list 1,99
  1  PROGRAM DebStruPtr ! Debug structures & pointers
  2      TYPE product
  3          INTEGER          id
  4          CHARACTER*16     name
  5          CHARACTER*8      model
  6          REAL             cost
  7          REAL             price
  8      END TYPE product
  9
 10      TYPE(product), TARGET :: prod1, prod2
 11      TYPE(product), POINTER :: curr, prior
 12
 13      curr => prod2
 14      prior => prod1
 15      prior%id = 82
 16      prior%name = "Schlepper"
 17      prior%model = "XL"
 18      prior%cost = 24.0
 19      prior%price = 104.0
 20      curr = prior
 21      WRITE ( *, * ) curr%name, " ", prior%name
 22  END PROGRAM DebStruPtr
(dbx) █

```

Example: Structures—set a breakpoint, and run under dbx.

The exact layout and messages may vary with each release.

```

(dbx) stop at 21
(1) stop at "DebStruc.f90":21
(dbx) run
Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f90"
  21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) █

```

Example: Structures—print an item of user-defined type.

```
(dbx) print prod1
prod1 = (
    id = 82
    name = "Schlepper "
    model = "XL "
    cost = 24.0
    price = 104.0
)
(dbx) ■
```

Above, dbx displays all fields of the user-defined type, including field names.

Example: Structures—inquire about an item of user-defined type.

Ask about the variable.

Ask about the type (-t).

```
(dbx) what is prod1
product prod1
(dbx) what is -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
end type product
(dbx) ■
```

Example: Structures—print a pointer, then quit dbx.

dbx displays the contents of a pointer, which is an address. This address can be different with every run.

```
(dbx) print prior
prior = (
    id    = 82
    name  = 'Schlepper'
    model = 'XL'
    cost  = 24.0
    price = 104.0
)
(dbx) quit
demo$ ■
```


Allocated Arrays

Example: Allocated arrays in dbx.

The exact layout and messages may vary with each release.

Alloc.f90

Unknown size is at line 6 →

Known size is at line 9 →

buffer(1000) holds 1000 →

```
demo% f90 -g Alloc.f90
demo% dbx a.out
(dbx) list 1,99
1  PROGRAM TestAllocate
2  INTEGER n, status
3  INTEGER, ALLOCATABLE :: buffer(:)
4  PRINT *, 'Size?'
5  READ *, n
6  ALLOCATE( buffer(n), STAT=status )
7  IF ( status /= 0 ) STOP 'cannot allocate buffer'
8  buffer(n) = n
9  PRINT *, buffer(n)
10 DEALLOCATE( buffer, STAT=status)
11 END
(dbx) stop at 6
(2) stop at "alloc.f90":6
(dbx) stop at 9
(3) stop at "alloc.f90":9
(dbx) run
Running: a.out
(process id 10749)
Size?
1000
stopped in main at line 6 in file "alloc.f90"
6          ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
continuing
stopped in main at line 7 in file "alloc.f90"
7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4  buffer(1:1000)
(dbx) cont
stopped in main at line 9 in file "alloc.f90"
9          PRINT *, buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
(dbx) ■
```

Print Arrays

Example: dbx recognizes arrays. It can print arrays.

```

demo$ dbx a.out
(dbx) list 1,25
Arraysdbx.f90
   1      DIMENSION iarr(4,4)
   2      DO i = 1,4
   3          DO j = 1,4
   4              iarr(i,j) = (i*10) + j
   5          END DO
   6      END DO
   7      END
(dbx) stop at 7
(1) stop at "Arraysdbx.f90":7
(dbx) run
Running: a.out
stopped in main at line 7 in file "Arraysdbx.f90"
   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

This section shows one way of printing portions of large arrays.

Example: dbx prints array *slices* if you specify which rows and columns.

```
ShoSli.f90 demo$ f90 -g ShoSli.f90
demo$ dbx 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.f90":7
(dbx) run
Running: a.out
stopped in main at line 7 in file "ShoSli.f90"
7          DO row = 1, 3
(dbx) ■
```

Example: Print row 3.


```
(dbx) print a(3:3,1:4)
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)
a(3:3, 1:4) =
      (1,4)   14
      (2,4)   24
      (3,4)   34
(dbx) ■
```

Generic Functions

Example: Generic function, cube root.

```
Generic.f90 (dbx) list 1,99
1  MODULE cr
2  INTERFACE cube_root
3  FUNCTION s_cube_root(x)
4  REAL :: s_cube_root
5  REAL, INTENT(IN) :: x
6  END FUNCTION s_cube_root
7  FUNCTION d_cube_root(x)
8  DOUBLE PRECISION :: d_cube_root
9  DOUBLE PRECISION, INTENT(IN) :: x
10 END FUNCTION d_cube_root
11 END INTERFACE
12 END MODULE cr
13 FUNCTION s_cube_root(x)
14 REAL :: s_cube_root
15 REAL, INTENT(IN) :: x
16 s_cube_root = x ** (1.0/3.0)
17 END FUNCTION s_cube_root
18 FUNCTION d_cube_root(x)
19 DOUBLE PRECISION :: d_cube_root
20 DOUBLE PRECISION, INTENT(IN) :: x
21 d_cube_root = x ** (1.0d0/3.0d0)
22 END FUNCTION d_cube_root
23 USE cr
24 REAL :: x, cx
25 DOUBLE PRECISION :: y, cy
26 WRITE(*, "('Enter a SP number: ')")
27 READ (*,*) x
28 cx = cube_root(x)
29 y = x
30 cy = cube_root(y)
31 WRITE(*, '("Single: ", F10.4, ", ", F10.4)') x, cx
32 WRITE(*, '("Double: ", F12.6, ", ", F12.6)') y, cy
33 WRITE(*, "('Enter a DP number: ')")
34 READ (*,*) y
35 cy = cube_root(y)
36 x = y
37 cx = cube_root(x)
38 WRITE(*, '("Single: ", F10.4, ", ", F10.4)') x, cx
39 WRITE(*, '("Double: ", F12.6, ", ", F12.6)') y, cy
40 END
```

Example: dbx with a generic function, cube root.

If asked "What is cube_root?",
dbx tells you there are two,
and asks you to select one.

If asked for cube_root(8)
dbx tells you there are two,
and asks you to select one.

If told to stop in cube_root,
dbx tells you there are two,
and asks you to select one.

From inside s_cube_root,
show current value of x.

```
(dbx) stop at 26
(2) stop at "Generic.f90":26
(dbx) run
Running: Generic
(process id 14633)
stopped in main at line 26 in file "Generic.f90"
    26      WRITE(*, "('Enter a SP number : ')")
(dbx) what is cube_root
More than one identifier 'cube_root.'
Select one of the following names:
    1) 'Generic.f90'cube_root s_cube_root ! real*4 s_cube_root
    2) 'Generic.f90'cube_root s_cube_root ! real*8 d_cube_root
> 1
real*4 function cube_root (x)
(dummy argument) real*4 x
(dbx) print cube_root(8.0)
More than one identifier 'cube_root.'
Select one of the following names:
    1) 'Generic.f90'cube_root ! real*4 s_cube_root
    2) 'Generic.f90'cube_root ! real*8 d_cube_root
> 1
cube_root(8) = 2.0
(dbx) stop in cube_root
More than one identifier 'cube_root.'
Select one of the following names:
    1) 'Generic.f90'cube_root ! real*4 s_cube_root
    2) 'Generic.f90'cube_root ! real*8 d_cube_root
> 1
(3) stop in cube_root
(dbx) cont
continuing
Enter a SP number:
8
stopped in cube_root at line 16 in file "Generic.f90"
    16      s_cube_root = x ** (1.0/3.0)
(dbx) print x
x = 8.0
(dbx) ■
```

Miscellaneous Tips

The following tips and background concepts can help.

Current Procedure and File

During a debug session, the Debugger 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.f90`, `a2.f90`, or `a3.f90`.

Uppercase Letters

In general, if your program has uppercase letters in any identifiers, then the Debugger recognizes them. You do not need to give it any specific *case sensitive/insensitive* commands, as in some earlier versions.

In fact, for `f90 1.0`, `f90` and `dbx` must both be in the case insensitive mode; that is, do *not* set “`dbxenv case sensitive`”.

Note – Names of *files* or *directories* are always case sensitive in both Debugger and `dbx`. This is true even if you have set the “`dbxenv case insensitive`” environment attribute.

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

Overview of dbx Features Useful for Fortran 90

The Debugger provides event management, process control, and data inspection. It allows you to watch what is happening during program execution.

With dbx, you can do such things as the following:

- *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)`

Help

At the Debugger prompt, to get:

- *All commands*—a list of commands, grouped by action, type `help`
- *Details of one command*—a command explanation, type `help cmdname`
- *Changes*—a list of the new and changed features, type `help changes`
- *FAQ*—answers to frequently asked questions, type `help FAQ`

Example: Command Summary (output varies with release).

```
(dbx) help
                        Command Summary
Execution and Tracing
cancel      catch      clear      cont      delete    fix
fixed      handler     ignore     intercept next      pop
replay     rerun      restore    run       save      status
step       stop      trace      unintercept when    whocatches
Displaying and Naming Data
assign     call       demangle   dis       display   down    dump
examine    exists    frame      hide      inspect   print   undisplay
unhide     up        whatis     where     whereami  whereis  which
<many commands omitted>
```

Example: “help *cmdnam*”—details for the where command.

```
(dbx) help where
where                # Print a procedure traceback
where <num>          # Print the <num> top frames in the traceback
where -f <num>       # Start traceback from frame <num>
where -h             # Include hidden frames
where -q            # Quick traceback (only function names)
where -v            # Verbose traceback (include function args and line
info)

Any of the above forms may be followed by a thread or LWP ID to obtain the
traceback for the specified entity.
(dbx) ■
```


Floating Point



This chapter is organized into the following sections.

<i>Summary</i>	<i>page 99</i>
<i>IEEE Solutions</i>	<i>page 101</i>
<i>The General Problems</i>	<i>page 100</i>
<i>IEEE Exceptions</i>	<i>page 102</i>
<i>IEEE Routines</i>	<i>page 103</i>
<i>Debugging IEEE Exceptions</i>	<i>page 113</i>
<i>Guidelines</i>	<i>page 113</i>
<i>Miscellaneous Examples</i>	<i>page 114</i>

8.1 Summary

This chapter introduces some floating-point issues. It focuses on IEEE floating point, and provides some reasons for it, some definitions, and some examples of how to use it. It lets you use IEEE floating point with some understanding. It is more tutorial than the other chapters, and deeper. This chapter is intended for scientists and engineers who use floating-point arithmetic in their work, but are not necessarily numerical analysts.

Read the *Numerical Computation Guide*, where you will find more complete explanations, examples, and details. You might also want to read *What Every Computer Scientist Should Know About Floating-point Arithmetic*, by David

Goldberg, which is in the on-line `READMEs` directory. It is a PostScript file and can be printed by `lpr` on any PostScript-compatible printer that has Palatino font. It can be viewed on-line by `pageview`.

8.2 *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 $A^T A$ 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 CDC¹, CRAY, and UNIVAC² 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.

1. CDC is a registered trademark of Control Data Corporation.

2. UNIVAC is a registered trademark of UNISYS Corporation.

8.3 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. IEEE offers choices, which the expert can use to good effect, while old arithmetic did it just one way.

What happens when we multiply two very large numbers with the same sign? Large numbers of different signs? Divide 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.

8.4 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()`” later in this chapter for details.

Default Signal Handlers

By default, `f90` 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.

Example: Get *no* default signal handlers (set `f77_no_handlers` to 1.)

1. Create the following C program.

```
demo$ cat NoHandlers.c
      int f90_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$ f90 NoHand.o Any.f90
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 – This variable is in the name space of the user's program, so do not use `f90_no_handlers` as the name of a variable anywhere else other than in the above C program.

8.5 IEEE Routines

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 `fpversion4`.

The the following interfaces help people use all the facets of IEEE arithmetic. These are mostly in the math library, in the `libm.a` and `libm.il` files, 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`

Flags and `ieee_flags()`

The `ieee_flags` function is part of the `libm` shipped with the operating system. 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 `ieee_flags` function can be used to query and 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 8-1 `ieee_flags` Argument Meanings

Value of <code>in</code> and <code>out</code>	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

Example: To determine what is the highest priority exception that has a flag raised, pass the input argument `in` as the null string.

```

ieeeer = 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.

```

ieeeer = ieee_flags( 'get', 'exception', 'overflow', out )
IF ( out.eq. 'overflow') PRINT *, 'overflow flag raised'

```

Example: Clear the invalid exception.

```

ieeeer = ieee_flags( 'clear', 'exception', 'invalid', out )

```

Example: Clear all exceptions.

```

ieeeer = ieee_flags( 'clear', 'exception', 'all', out )

```

Example: Set rounding direction to zero.

```
ieeeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

Example: Set rounding precision to double.

```
ieeeer = 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 )
```

Calls of this form are used in the next two examples.

Values and `ieee_values()`

The `ieee_values(3m)` file is a collection of functions. Each function returns a special IEEE value. The Fortran names for these functions are listed in `libm_double(3f)` and `libm_single(3f)`. You can use special IEEE entities, such as *infinity* or *minimum normal*, in a user program. See also the man page `ieee_values(3m)`.

Example: A convergence test might be like this.

```
IF ( delta .LE. r_min_normal() ) RETURN
```


The IEEE values available are listed in the table below.

Table 8-2 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)`; see `libm_double(3f)`, `libm_single(3f)`, and `ieee_functions(3m)`.

Exception Handlers and `ieee_handler()`

A floating-point user may 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 information about an exception:

- Generate a signal for a floating-point exception. The official UNIX name for signal is *floating-point exception* is `SIGFPE`.
- To generate a `SIGFPE`, establish a signal handler. The default behavior on SPARC hardware systems is “do *not* generate a `SIGFPE`.”

Establishing a Signal Handler Function with `ieee_handler()`

To establish a signal handler, pass the following to `ieee_handler()`:

- Name of the function
- Exception to watch for
- Action to take

Once a handler is established, a signal is generated whenever the particular floating-point exception occurs.

The form of invoking `ieee_handler()` is as follows.

<code>i = ieee_handler(action, exception, handler)</code>		
<i>action</i>	character	"get" or "set" or "clear"
<i>exception</i>	character	"invalid" or "division" or "overflow" or "underflow" or "inexact"
<i>handler</i>	function name	The name of the function you wrote
return value	integer	0=OK

Writing a Signal Handler Function

Actions taken by the function are up to you, but the *form* of the function is:

- The function must be an integer function.
- The function must have three arguments, typed as follows:

Pattern— `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

Form: Signal handler function.

5.x

SunOS 5.x Form

```

INTEGER FUNCTION hand5x( sig, sip, uap)
INTEGER sig, location
TYPE fault_typ
    INTEGER address
END TYPE fault_typ
TYPE siginfo
    INTEGER si_signo
    INTEGER SI_CODE
    INTEGER si_errno
    TYPE(fault_typ) fault
END TYPE siginfo
TYPE(siginfo) sip
location = sip%fault%address
... actions you take ...
END

```

Pattern— hand4x(sig, code, context)

hand4x is your name for your integer functionForm: Signal handler function.
 Example: Detect an exception using a handler, and abort—SunOS 5.x or 4.x.

5.x and 4.x

DetExcHan.f90
 SunOS 5.x or 4.x

SIGFPE is generated whenever a
 floating-point exception occurs→

Then the SIGFPE is detected and
 control is passed to the
 myhandler function.

```

PROGRAM DetExcH
EXTERNAL myhandler
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.x as all we do is abort
INTEGER sig, code
INTEGER, DIMENSION(5) :: context
CALL abort()
END
demo% f90 DetExcHan.f90
demo% a.out
abort: called
Abort (core dumped)
demo% █

```

Example: Locate an exception (get address) using a handler

5.x

LocExcHan5x.f90

SunOS 5.x

An **address** is mostly for those who use such low-level debuggers as adb.

For how get the **line number** see Section 8.6, "Debugging IEEE Exceptions" for details.

Caveat I/O in a handler is risky. Calling abort() reduces the risk.

```
PROGRAM LocExcH5x ! Locate Exception, by Handler - SunOS 5.x
  EXTERNAL hand5x
  INTEGER hand5x
  REAL :: r = 14.2, s = 0.0
  i = ieee_handler('set', 'division', hand5x)
  t = r/s
END
INTEGER FUNCTION hand5x(sig, sip, uap) ! Handler - SunOS 5.x
  INTEGER location, sig
  TYPE fault_typ
    INTEGER address
  END TYPE fault_typ
  TYPE siginfo
    INTEGER si_signo
    INTEGER si_code
    INTEGER si_errno
    TYPE(fault_typ) fault
  END TYPE siginfo
  TYPE(siginfo) sip
  location = sip%fault%address
  WRITE (*, "('Exception at ',Z8)") location ! Risky in a handler
  CALL abort() ! Just to reduce risk.
END
```

Example: Locate an exception (get address) using a handler.

5.x

Compile/Load/Run
SunOS 5.x

The actual address varies with installation and architecture.

```
demo% f90 LocExcHan5x.f90
demo% a.out
Exception at 11FA8
abort: called
Abort (core dumped)
Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M): Division by Zero;
Sun's implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
demo% █
```

Example: Locate an exception (get address) using a handler

4.x

Compile/Load/Run
SunOS 4.xThe actual address varies with
installation and architecture.

```
demo% f90 LocExcHan4x.f90
demo% a.out
Exception at pc 9172
abort: called
Abort (core dumped)
Note: Following IEEE floating-point traps enabled;see
ieee_handler(3M): Division by Zero;
Sun's implementation of IEEE arithmetic is discussed in
the Numerical Computation Guide.
demo% █
```

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 90, this function is called automatically just before execution terminates.

The message typically looks like this (may vary slightly 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 90, 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 underflow and inexact exceptions, you will get a message if any two of the underflow and inexact exceptions occur, in general.

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. Clearing all messages is not recommended.

You can gain complete control with `ieee_handler()`. In your own exception handler routine you can specify actions, and you can turn off messages with `ieee_flags()` by clearing exception status flags.

8.6 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
```

You need to do these two things:

- Establish a signal handler, using `ieee_handler(3m)`.

This will cause a SIGFPE to be generated when a floating-point exception occurs.

- After you invoke `dbx`, enter the “catch FPE” command.

This causes `dbx` to listen for any SIGFPE, and halt when it hears one. See the subsection on where the exception occurred in Section 7.2, “The `dbx` Debugger” for explicit examples.

8.7 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`.

8.8 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 $1.0\text{E-}38$ 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 $1.40130\text{E-}45$. 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 90 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. 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 $\pm\text{Inf}$ 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. 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.

Example: Excessive underflow.

```
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:

```
real 0m4.50s
user 0m0.11s
sys 0m4.35s
```

So the real computation required about 0.1 seconds, but the software fix required four seconds. In a real application this can be *hours*. Clearly this is not desirable.

Solution 1: Change All of the Program

If you change the code to be double precision (by rewriting the code with double precision variables) you get vast improvement.

```
real 0m0.20s
user 0m0.08s
sys 0m0.11s
```

Now, of course, 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

If you declare `sum` to be `DOUBLE PRECISION`, and change only the summation line of code as follows:

```
sum = sum + a(i)*dble(b(i))
```

then you get

```
real 0m0.18s
user 0m0.06s
sys  0m0.11s
```

By promoting one variable to double, you eliminate the software underflow problem. Note that in a real application, 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 like this.

```
real 0m0.18s
user 0m0.01s
sys  0m0.13s
```

Note that this time is very nearly 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.

C–Fortran Interface



This chapter is organized into the following sections.

<i>Sample Interface</i>	<i>page 120</i>
<i>How to Use this Chapter</i>	<i>page 121</i>
<i>Compatibility Requirements</i>	<i>page 122</i>
<i>Fortran Calls C</i>	<i>page 134</i>
<i>C Calls Fortran</i>	<i>page 149</i>

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

Purpose—This chapter shows how to write Fortran 90 routines that call C routines, and C routines that call Fortran 90 routines. A common reason to do such calls is to use existing libraries.

Caveat—This subject requires more sophistication than most of this manual. To paraphrase Hotspur, *any programmer can write such programs, but will they work when you do call for them?*

Approach—This chapter lists the compatibility rules and shows examples for item passable between C and Fortran. Examples show *how to pass an item*, not how it would be used in real applications.

9.1 Sample Interface

Example: A C function to be called by a Fortran main program.

```
Samp.c      samp_ ( int *i, float *f ) /* both i and f are pointers */
{
    *i = 9;
    *f = 9.9;
}
```

Example: A Fortran main program to call a C function.

```
Sampmain.f90  PROGRAM Sample
               INTEGER i
               REAL r
               CALL Samp ( i, r ) ! both i and r are passed by reference
               WRITE( *, "(I2, F4.1)") i, r
               END PROGRAM Sample
```

Example: A Fortran program calls a C function.

Compile/Link/Execute.

```
demo$ cc -c Samp.c
demo$ f90 Samp.o Sampmain.f90 ← This does the linking
demo$ a.out
 9 9.9
demo$ █
```

9.2 How to Use this Chapter

1. Examine the previous section, “Sample Interface.”
2. Examine the next section, “Compatibility Requirements.”
3. Find what to do in the section “Fortran Calls C” or “C Calls Fortran.”

The following two tables help find the appropriate subsection.

For a Fortran 90 main and a C function:

<i>Fortran Calls C</i>	<i>page 134</i>
<i>Arguments Passed by Reference (f90 Calls C)</i>	<i>page 134</i>
<i>Simple Arguments Passed by Reference (f90 Calls C)</i>	<i>page 134</i>
<i>Complex Arguments Passed by Reference (f90 Calls C)</i>	<i>page 135</i>
<i>Character Arguments Passed by Reference (f90 Calls C)</i>	<i>page 136</i>
<i>Vector Arguments Passed by Reference (f90 Calls C)</i>	<i>page 138</i>
<i>Matrix Arguments Passed by Reference (f90 Calls C)</i>	<i>page 139</i>
<i>Structure Arguments Passed by Reference (f90 Calls C)</i>	<i>page 140</i>
<i>Pointer Arguments Passed by Reference (f90 Calls C)—N/A</i>	<i>page 140</i>
<i>Arguments Passed by Value (f90 Calls C)</i>	<i>page 141</i>
<i>Function Return Values (f90 Calls C)</i>	<i>page 141</i>
<i>INTEGER Function Return Value (f90 Calls C)</i>	<i>page 141</i>
<i>REAL Function Return Value (f90 Calls C)</i>	<i>page 142</i>
<i>Pointer-to-a-REAL Function Return Value (f90 Calls C)</i>	<i>page 143</i>
<i>DOUBLE PRECISION Function Return Value (f90 Calls C)</i>	<i>page 144</i>
<i>LOGICAL Function Return Value (f90 Calls C)</i>	<i>page 145</i>
<i>CHARACTER Function Return Value (f90 Calls C) N/A</i>	<i>page 146</i>
<i>Labeled Common (f90 Calls C)</i>	<i>page 147</i>
<i>Alternate Returns (f90 Calls C) - N/A</i>	<i>page 148</i>

For a C main and a Fortran 90 subprogram:)

<i>C Calls Fortran</i>	<i>page 149</i>
<i>Arguments Passed by Reference (C Calls f90)</i>	<i>page 149</i>
<i>Simple Arguments Passed by Reference (C Calls f90)</i>	<i>page 149</i>
<i>Complex Arguments Passed by Reference (C Calls f90)</i>	<i>page 150</i>
<i>Character Arguments Passed by Reference (C Calls f90)</i>	<i>page 151</i>
<i>Vector Arguments Passed by Reference (C Calls f90)</i>	<i>page 152</i>
<i>Matrix Arguments Passed by Reference (C Calls f90)</i>	<i>page 152</i>
<i>Structure Arguments Passed by Reference (C Calls f90)</i>	<i>page 153</i>
<i>Pointer Arguments Passed by Reference (C Calls f90)—N/A</i>	<i>page 155</i>
<i>Arguments Passed by Value (C Calls f90) - N/A</i>	<i>page 155</i>
<i>Function Return Values (C Calls f90)</i>	<i>page 155</i>
<i>INTEGER Function Return Value (C Calls f90)</i>	<i>page 155</i>
<i>REAL Function Return Value (C Calls f90)</i>	<i>page 156</i>
<i>DOUBLE PRECISION Function Return Value (C Calls f90)</i>	<i>page 156</i>
<i>LOGICAL Function Return Value (C Calls f90)</i>	<i>page 158</i>
<i>CHARACTER Function Return Value (C Calls f90)</i>	<i>page 159</i>
<i>Labeled Common (C Calls f90)</i>	<i>page 160</i>
<i>Alternate Returns (C Calls f90)</i>	<i>page 161</i>

9.3 Compatibility Requirements

Most C/Fortran interfaces must get all of these aspects right:

- Function or subroutine
- Underscore in names of routines
- Upper and lowercase in identifiers
- Data type compatibility
- Passing arguments by reference or value
- String arguments and order
- 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.

Underscore in Names of Routines

The Fortran compiler 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.

Each subprogram name must have 31 or fewer characters.

To avoid the underscore problem, in the C function definition, change the name of the C function by appending an underscore to that name.

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 identifiers to lowercase. It converts all uppercase letters to lowercase letters, except within character-string constants.

To avoid the case sensitivity problem—in the C function definition, make the name of the C function all lowercase.

Data Type Compatibility

You may want to write Fortran 90 routines to interface with existing C routines, or C routines to interface with existing Fortran 90 routines.

Writing Fortran 90 Code for Existing C Routines

For any given C intrinsic data type, the following table provides a close corresponding Fortran 90 data type.

Table 9-1 C Data Type to Fortran 90 Data Type

<i>C Intrinsic Type</i>	<i>Close Fortran 90 Type</i>	<i>Size (Bytes)</i>	<i>Alignment (Bytes)</i>
<code>char x ;</code>	CHARACTER x	1	1
<code>signed char x ;</code>	INTEGER (KIND=1) x	1	4
<code>signed char x[n] ;</code>	CHARACTER (LEN=n) x	n	1
<code>unsigned char x[n] ;</code>	CHARACTER (LEN=n) x	n	1
<code>float x ;</code>	REAL x	4	4
<code>double x ;</code>	DOUBLE PRECISION x	8	4
<code>long double x ;</code>	N/A	16	4
<code>int x ;</code>	INTEGER x	4	4
<code>signed x ;</code>	INTEGER x	4	4
<code>signed int x ;</code>	INTEGER x	4	4
<code>long x ;</code>	INTEGER x	4	4
<code>long int x ;</code>	INTEGER x	4	4
<code>signed long x ;</code>	INTEGER x	4	4
<code>signed long int x ;</code>	INTEGER x	4	4
<code>unsigned int x ;</code>	INTEGER x	4	4
<code>unsigned long x ;</code>	INTEGER x	4	4
<code>unsigned long int x ;</code>	INTEGER x	4	4
<code>short x ;</code>	INTEGER (KIND=2) x	2	4
<code>short int x ;</code>	INTEGER (KIND=2) x	2	4
<code>signed short int x ;</code>	INTEGER (KIND=2) x	2	4
<code>unsigned short x ;</code>	INTEGER (KIND=2) x	2	4
<code>unsigned short int x ;</code>	INTEGER (KIND=2) x	2	4
<code>long long x ;</code>	N/A	-	-
<code>unsigned long long x ;</code>	N/A	-	-

C double aligns on 8-byte boundaries, unless in a common block, then on 4.

Writing C Code for Existing Fortran 90 Routines

For any given Fortran 90 intrinsic data type, the following table provides a close corresponding C data type.

Table 9-2 Fortran 90 Data Type to C Data Type

<i>Fortran 90 Intrinsic Data Type</i>	<i>Close C Data Type</i>	<i>Size (Bytes)</i>	<i>Alignment (Bytes)</i>
CHARACTER x	unsigned char x ;	1	1
CHARACTER (LEN=n) x	unsigned char x[n] ;	n	1
CHARACTER (LEN=n, KIND=1) x	unsigned char x[n] ;	n	1
COMPLEX x	struct {float r,i;} x;	8	4
COMPLEX (KIND=4) x	struct {float r,i;} x;	8	4
COMPLEX (KIND=8) x	struct {double dr,di;} x;	16	4
DOUBLE PRECISION x	double x ;	8	4
REAL x	float x ;	4	4
REAL (KIND=4) x	float x ;	4	4
REAL (KIND=8) x	double x ;	8	4
INTEGER x	int x ;	4	4
INTEGER (KIND=1) x	signed char x ;	1	4
INTEGER (KIND=2) x	short x ;	2	4
INTEGER (KIND=4) x	int x ;	4	4
LOGICAL x	int x ;	4	4
LOGICAL (KIND=1) x	signed char x ;	1	4
LOGICAL (KIND=2) x	short x ;	2	4
LOGICAL (KIND=4) x	int x ;	4	4

In the current release, with items of type `INTEGER` for `KIND=1, 2, or 4`:

- Each uses 4 bytes of storage
- Each aligns on 4-byte boundaries
- Each involves 32 bits if any computations are involved

Passing Arguments by Reference or Value

C and Fortran 90 pass arguments using the following different basic rules.

- Fortran 90 generally passes arguments by reference.
- C passes arguments by value.

Despite this seeming conflict, some compatibility can be achieved, at least for the Fortran 90 pass by reference. The compatibility is possible because a C program can specify that the value being passed is actually an *address*. For C, this is traditionally described as *passing pointers*.

Passing Arguments by Reference

Pass by reference in Fortran 90 can be made compatible with *pass pointers* in C.

In C, *passing pointers* is specified in either of the following ways—one for defining new functions, and another for invoking existing functions.

- Defining New Functions

In a function, where data types of arguments are declared, if you precede a dummy argument by an asterisk (*), C passes a pointer to the item.

Example: Define a C function—a dummy argument is a pointer to an `int`.

```
void simrefl_ ( int * d )    /* d is a pointer to an int */
{
    ...
}
```

- Invoking Existing Functions

To make C pass the address of the argument, in the statement that invokes the function do the following:

- If the item is not a character string or array, then precede the actual argument by an ampersand (&).

Example: An actual argument is a pointer to an int.

```
int a ;
...
simref2_ ( &a )    /* &a is a pointer to an int */
...
```

- If the item is a character string or array, then do *not* precede the actual argument by an ampersand (&).

Example: An actual argument is a pointer to a character string.

```
char s[9] ;
...
simref3_ ( s )    /* s is a pointer to a character string */
...
```

C always passes arrays and character strings using pointers. In fact, C *universally* promotes character strings and arrays to pointers. So in C, you cannot pass arrays or character strings by value.

Passing Arguments by Value (N/A)

If a C function passes an argument by value *using no pointers*, there is no compatible way of passing to Fortran 90. That is, Fortran 90, cannot interface with such a function.

Example: Define a C function—dummy argument is an int (no pointer).

```
simvall_ ( int i )
{
    ...
}
```

The above function cannot be called from Fortran 90.

Character Strings and Order

Passing strings between C and Fortran is nonstandard. It is not encouraged.

The following compatibility rules are provided for those who need to do it, despite the complexities.

Rules for Passing Any Character Strings

- If you make a string in Fortran and pass it to C, you must provide an explicit null terminator because Fortran does not automatically do that, and C expects (requires) it.
- All C character strings pass using pointers.

Arguments that are Character Strings

- For a character *argument*, Fortran 90 passes/needs an extra argument that:
 - Contains the length of the character string
 - Is equivalent to a C `long int`
 - Is passed by value
- The order of arguments is as follows.
 - A list of the regular arguments
 - A list of hidden arguments, one for each character string argument

The list of hidden arguments comes after the list of regular arguments.

- For Fortran 90 calling C, if Fortran 90 passes a character string argument, the C function can ignore the extra argument or use it.
- For C calling Fortran 90, C *must* pass the extra argument because Fortran 90 expects (requires) it.

Example: Fortran string argument, passed by reference—a Fortran call.

```
CHARACTER*7 s
INTEGER b
...
CALL sam ( s, b )
```

Example: The above Fortran call is equivalent to the following C call.

```
char s[7] ;
long b ;
...
sam_ ( s, &b, 7L ) ;
```

In the above example:

- `s` is passed by pointer because `s` is a character string.
- `b` is passed by pointer because we explicitly use an ampersand (&).
- `7`, the length, is passed by value (without a pointer) as a literal `7 long`.

Functions that are Character Strings

The *returned* character string is passed as two extra initial *arguments*, in the following order:

- A pointer to the start of the string return value
- The length of the string return value

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 and higher arrays, switch subscripts in all references and declarations.

Tip

Many users tell us that it gets confusing, say, to triangularize in C and then pass the parts to Fortran. More generally, it may be confusing to do some of the matrix manipulation in C and some in Fortran. So if passing parts of arrays between C and Fortran does not work (or if it is confusing), try passing the *whole* array to the other language and do *all* the matrix manipulation there; avoid doing part in C and part in Fortran.

Libraries and Linking with the `f90` Command

To get the proper Fortran libraries linked, use the `f90` 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 `f90` to link.

```
demo$ f90 -c RetCmplx.f90
demo$ cc -c RetCmplxmain.c
demo$ f90 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$ f90 -c RetCmplx.f90
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 9-3 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 <code>OPEN(3S)</code> .	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.

9.4 Fortran Calls C

Arguments Passed by Reference (f90 Calls C)

Simple Arguments Passed by Reference (f90 Calls C)

The simple arguments are of Fortran 90 data types INTEGER, REAL, DOUBLE PRECISION, or LOGICAL, without pointers, dimensions, or structures.

Example: Simple arguments, C function—C arguments as pointers.

```
SimRef.c void simref_ ( b4, i4, r4, d8 )
/* Simple types, passed by reference, from f90 (f90 calls C)*/
int      * b4 ;
int      * i4 ;
float    * r4 ;
double   * d8 ;
{
    *b4 = 1 ;
    *i4 = 9 ;
    *r4 = 9.9f ;
    *d8 = 9.9F ;
}
```

Example: Simple arguments—Fortran default way.

```
SimRefmain.f90 PROGRAM SimpleRef
! Pass some simple types, by reference, to C (f90 calls C)
LOGICAL      b4 ! Default kind is 4-byte
INTEGER      i4 ! Default kind is 4-byte
REAL         r4 ! Default kind is 4-byte
DOUBLE PRECISION d8! This is 8-byte
CALL SimRef ( b4, i4, r4, d8 )
WRITE( *, '(L4,I4,F6.1,F6.1)' ) b4, i4, r4, d8
END PROGRAM SimpleRef
```

Example: Simple arguments with Fortran and C.

Compile/Link/Execute

```
demo$ cc -c SimRef.c
demo$ f90 SimRef.o SimRefmain.f90
demo$ a.out
      T    9    9.9    9.9
demo$ █
```

Complex Arguments Passed by Reference (f90 Calls C)

Example: Complex arguments, C function—pointers to structures.

CmplxRef.c

```
void cmplxref_ ( struct complex { float r, i; } *w
                struct dcomplex { double r, i; } *z )
{
    w -> r = 6;
    w -> i = 7;
    z -> r = 8;
    z -> i = 9;
}
```

Example: Complex arguments, Fortran main program calls C function.

CmplxRefmain.f90

```
PROGRAM ComplexRef
! Pass complex types, by reference, to C (f90 calls C)
INTEGER, PARAMETER :: doublecomplex=8
COMPLEX w
COMPLEX (KIND=doublecomplex) z
CALL CmplxRef ( w, z )
WRITE(*,*) w
WRITE(*,*) z
END PROGRAM ComplexRef
```

Example: Complex arguments.

Compile/Link/Execute

```
demo$ cc -c CmplxRef.c
demo$ f90 CmplxRef.o CmplxRefmain.f90
demo$ a.out
( 6.0000000, 7.0000000)
( 8.0000000000000000, 9.0000000000000000)
demo$ █
```

Character Arguments Passed by Reference (F90 Calls C)

Passing strings between C and Fortran is nonstandard. It is not encouraged.

For Fortran 90 calling C, if Fortran 90 passes a character string argument, it always passes an extra, hidden argument. The C function can ignore these extra arguments or it can use them. For the detailed requirements of passing string arguments, see “Character Strings and Order” on page 129.

Ignoring Extra Arguments for Strings

A C function can *ignore* the extra arguments, since they are *after* the list of other arguments.

Example: Character arguments—this C function *ignores* the extra arguments.

StrRefI.c

```
void strrefi_ ( char *a, char *z )
{
    static char ax[11] = "abcdefghij" ;
    static char zx[31] = "abcdefghijklmnopqrstuvwxyz" ;
    strncpy ( a, ax, 10 ) ;
    strncpy ( z, zx, 26 ) ;
}
```

Example: Character arguments—a Fortran call passes hidden extra arguments.

StrRefImain.f90

```
PROGRAM StringRefI
  CHARACTER a*10, z*30
  a = ' '
  z = ' '
  CALL StrRefI( a, z )
  WRITE (*, 1) a, z
1  FORMAT("a='", A, "'", /, "z='", A, "'")
END PROGRAM StringRefI
```

Example: Character arguments, Fortran and C, C *ignores* the extra arguments.

Compile/Link/Execute

```
demo$ cc -c StrRefI.c
demo$ f90 StrRefI.o StrRefmain.f90
demo$ a.out
s10='abcdefghij'
s30='abcdefghijklmnopqrstuvwxyz'
demo$ █
```

Using Extra Arguments for Strings

A C function can *use* the extra arguments from Fortran; they are *after* the list of other arguments, and they are passed *without* pointers.

Example: Character arguments—a C function *uses* the extra arguments.

```
StrRefU.c      strrefu_ ( char *a, char *z, long L10, long L30 )
{
    static char ax[11] = "abcdefghij" ;
    static char zx[31] = "abcdefghijklmnopqrstuvwxyz" ;

    printf("%d %d \n", L10, L30 ) ;      /* Use L10 and L30, print them */
    strncpy ( a, ax, 11 ) ;
    strncpy ( z, zx, 26 ) ;
}
```

Example: Character arguments—a Fortran call makes hidden extra arguments.

```
StrRefUmain.f90  PROGRAM StringRefU
                  CHARACTER a*10, z*30
                  a = ' '
                  z = ' '
                  CALL StrRefU( a, z )
                  WRITE (*, 1) a, z
1  FORMAT("a=' ", A, "'", /, "z=' ", A, "'")
                  END PROGRAM StringRefU
```

Example: Character arguments, Fortran and C, C *uses* the extra arguments.

Compile/Link/Execute

```
10 30
s10='abcdefghij'
s30='abcdefghijklmnopqrstuvwxyz'
```

The above C function prints the extra arguments (the lengths); what you really do with them is up to you

Vector Arguments Passed by Reference (f90 Calls C)

Example: A C one-dimensional array argument, indexed from 0 to 8.

```
FixVec.c
void fixvec_ ( int v[9], int *sum )
{
    int i;
    *sum = 0;
    for ( i = 0; i <= 8; i++ ) *sum = *sum + v[i];
}
```

Example: A Fortran vector argument, implicitly indexed from 1 to 9.

```
FixVecmain.f90
PROGRAM FixedVector
    INTEGER, DIMENSION(9) :: a = (/ 1,2,3,4,5,6,7,8,9 /)
    INTEGER i, sum
    CALL FixVec ( a, sum )
    WRITE(*, '("a: ", 9I2, ", sum:" I3)') (a(i),i=1,9), sum
END PROGRAM FixedVector
```

Example: A vector argument, Fortran and C, implicitly indexed from 1 to 9.

```
Compile/Link/Execute
demo$ cc -c FixVec.c
demo$ f90 FixVec.o FixVecmain.f90
demo$ a.out
a:  1 2 3 4 5 6 7 8 9, sum: 45
demo$ █
```

Example: A vector argument, Fortran and C, explicitly indexed from 0 to 8.

```
FixVecmainE.f90
PROGRAM FixedVectorE ! a is explicitly indexed 0 to 8
    INTEGER, DIMENSION(0:8) :: a = (/ 1,2,3,4,5,6,7,8,9 /)
    INTEGER i, sum
    CALL FixVec ( a, sum )
    WRITE(*, '("a: ", 9I2, ", sum:" I3)') (a(i),i=0,8), sum
END PROGRAM FixedVectorE
```

Example: A vector argument, Fortran and C, explicitly indexed from 0 to 8.

```
Compile/Link/Execute
demo$ cc -c FixVec.c
demo$ f90 FixVec.o FixVecmain#.f90
demo$ a.out
a:  1 2 3 4 5 6 7 8 9, sum: 45
demo$ █
```


Matrix Arguments Passed by Reference (f90 Calls C)

In a two-dimensional array, the rows and columns are switched.

Example: A 2 by 2 C array argument, indexed from 0 to 1.

```
FixMat.c      fixmat_ ( int a[2][2] )
               {
                   a[0][1] = 99 ;    /* C changes a[0][1] */
               }
```

Example: A 2 by 2 Fortran array argument, explicitly indexed from 0 to 1.

```
FixMatmain.f90  PROGRAM FixedMatrix
                  INTEGER c, r
                  INTEGER, DIMENSION(0:1,0:1) :: m
                  m(0,0)=00 ; m(0,1)=01 ; m(1,0)=10 ; m(1,1)=11
                  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 PROGRAM FixedMatrix
```

Example: A 2 by 2 array argument—show *m* before and after the C call.

Compile/Link/Execute

Before →

After →

```
demo$ cc -c FixMat.c
demo$ f90 FixMat.o FixMatmain.f90
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$ █
```

←Fortran shows that *m*(1,0) got changed

Structure Arguments Passed by Reference (f90 Calls C)

Example: A C structure argument—an integer and a character string.

```
StruRef.c
struct InCh {           /* Define a structure */
    int nbytes ;
    char a[16] ;
} ;

void struref_ ( v )     /* Use the structure in definining a function */
struct InCh *v ;
{
    bcopy( "oyvay", v->a, 5 ) ; /* Change the char component */
    v -> nbytes = 5 ;          /* Change the int component */
}
```

Example: A Fortran 90 structure argument, an integer and a character string.

```
StruRefmain.f90
PROGRAM StructureRef
    TYPE IntChr           ! Define the derived type IntChr
        INTEGER    n
        CHARACTER  str*15
    END TYPE IntChr

    TYPE(IntChr) vls      ! Make vls an item of type IntChr

    vls % n = 0           ! Initialize components
    vls % str = '123456789012345'
    CALL StruRef ( vls )  ! Change components
    WRITE ( *, 1 ) vls % n, vls % str ! Print components
1  FORMAT("n =", I2, ", str='", A, "'")
END PROGRAM StructureRef
```

Example: A structure argument, Fortran 90 and C.

```
Compile/Link/Execute
demo$ cc -c StruRef.c
demo$ f90 StruRef.o StruRefmain.f90
demo$ a.out
n = 5, str='oyvay6789012345'
demo$ █
```

Pointer Arguments Passed by Reference (f90 Calls C)—N/A

These two kinds of pointers are not compatible.

Arguments Passed by Value (f90 Calls C)

In general, if an existing C function passes *arguments* by value using no pointers, then Fortran 90 cannot use that function.

Function Return Values (f90 Calls C)

In general, for a C *function return value*:

- If it has no pointer, then Fortran 90 can use it as is.
- If it returns a pointer, then Fortran 90 needs an interface block. See “Pointer-to-a-REAL Function Return Value (f90 Calls C)” on page 143.

INTEGER Function Return Value (f90 Calls C)

Example: A C function with an `int` function return value (not a pointer).

```
RetInt.c
int retint_ ( int *r )
{
    int s ;
    s = *r ;
    s++ ;
    return ( s ) ;
}
```

Example: A Fortran program uses a C function that returns an `int`

```
RetIntmain.f90
PROGRAM ReturnInt
  INTEGER r, s, RetInt
  r = 8
  s = 100 + RetInt ( r ) ! The C function is invoked here
  WRITE( *, "(2I4)") r, s
END PROGRAM ReturnInt
```

Example: Fortran and C with an `INTEGER` function return value.

Compile/Link/Execute

```
demo$ cc -c RetInt.c
demo$ f90 RetInt.o RetIntmain.f90
demo$ a.out
      8 109
demo$
```

REAL *Function Return Value* (f90 Calls C)

Example: A C function yields a float function return value (not a pointer).

```
RetFloat.c  float  retfloat ( float *pf )
             {
               float  f ;
               f = *pf ;
               f++ ;
               return ( f ) ;
             }
```

Example: A Fortran program uses a C function that returns a float.

```
RetFloatmain.f90  PROGRAM ReturnFloat
                   REAL  RetFloat, r, s
                   r = 8.0
                   s = 100.0 + RetFloat ( r ) ! The C function is invoked here
                   WRITE(*, '(2F6.1)' )  r, s
                   END PROGRAM ReturnFloat
```

Example: Fortran and C with a REAL function return value.

```
Compile/Link/Execute  demo$ cc -c RetFloat.c
                     demo$ f90 RetFloat.o RetFloatmain.f
                     demo$ a.out
                        8.0 109.0
                     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.

Pointer-to-a-REAL Function Return Value (F90 Calls C)

In general, if an existing C function returns a pointer to an item, then Fortran 90 requires an interface block for the function.

Example: A C function with a pointer-to-a-float function return value.

```
RetPtrF.c
static float f;
float *retptrf ( float *a )
{
    f = *a ;
    f++ ;
    return &f ;
}
```

Example. A Fortran program uses a pointer-to-a-REAL function return value.

```
RetPtrFmain.f90
PROGRAM ReturnPtrFloat
! Use a C function return value that is a pointer to a real.
INTERFACE
    FUNCTION RetPtrF ( x )
        REAL          x
        REAL, POINTER :: RetPtrF
    END FUNCTION RetPtrF
END INTERFACE
REAL a, b
a = 8.0
b = 100.0 + RetPtrF(a) ! Uses C function here
WRITE(*,'(F9.0)') b
END PROGRAM ReturnPtrFloat
```

Example: Fortran and C with a pointer-to-a-REAL function return value.

Compile/Link/Execute

```
demo$ cc -c RetPtrF.c
demo$ f90 RetPtrF.o RetPtrFmain.f90
demo$ a.out
109.
demo$ █
```

DOUBLE PRECISION *Function Return Value* (f90 Calls C)

Example: A C function with a double function return value (not a pointer).

```
RetDbl.c
double retdbl_ ( double *r )
{
    double s ;
    s = *r ;
    s++ ;
    return ( s ) ;
}
```

Example: Fortran 90 uses a DOUBLE PRECISION function return value from C.

```
RetDblmain.f90
PROGRAM ReturnDbl
    DOUBLE PRECISION r, s, RetDbl
    r = 8.0
    s = 100.0 + RetDbl(r) ! The C function is invoked here
    WRITE( *, "(2F6.1)") r, s
END PROGRAM ReturnDbl
```

Example: Fortran and C with a DOUBLE PRECISION function return value.

Compile/Link/Execute

```
demo$ cc -c RetDbl.c
demo$ f90 RetDbl.o RetDblmain.f90
demo$ a.out
      8.0 109.0
demo$ █
```

LOGICAL *Function Return Value* (f90 Calls C)

Example: A C function with an `int` function return value (not a pointer).

```
RetLog.c  int retlog_ ( int *r )
          {
            int s;
            s = *r;
            if ( s == 0 ) s = 1 ; else s = 0 ;
            return ( s );
          }
```

Example: A Fortran program uses a C function as if it returns a LOGICAL.

```
RetLogmain.f90  PROGRAM TryRetLog
                 LOGICAL r, s, RetLog
                 r = .FALSE.
                 s = .TRUE. .AND. RetLog(r)
                 WRITE( *, "(2L4)") r, s
                 END PROGRAM TryRetLog
```

Example: Fortran and C with a LOGICAL function return value.

Compile/Link/Execute

```
demo$ cc -c RetLog.c
demo$ f90 RetLog.o RetLogmain.f90
demo$ a.out
      F   T
demo$ █
```

CHARACTER *Function Return Value* (f90 Calls C) N/A

Passing character strings between C and Fortran is not encouraged.

See “Character Strings and Order” on page 129, for details of compatibility.

Example. A C character string function for Fortran.

RetStr.c

The function value is passed not as a “*function return value*” but as these *arguments*:

rval_ptr, pointer to string
rval_len, length of string

The normal string argument is passed as:

&ch_ptr, pointer to string
ch_len, length of string

```
void retstr_ ( char *rval_ptr, /* pointer to returned string */
              int rval_len,   /* length of returned string */
              char *ch_ptr,   /* pointer to string argument */
              int *n_ptr,     /* pointer to number of copies */
              int ch_len )    /* length of string argument */
{ /* Return string: n_ptr copies of the character ch_ptr */
  int count, i ;
  char *cp ;
  count = *n_ptr ;
  cp = rval_ptr ;
  for (i=0; i<count; i++) {
    *cp++ = *ch_ptr ;
  }
}
```

Example. A Fortran program uses a C CHARACTER function.

RetStrmain.f90

```
PROGRAM TryRetStr
  CHARACTER String*16, RetStr*9
  String = ' '
  String = '1234' // RetStr(' ',9) // '456' ! Use C function here
  WRITE(*,*) '"', String, '"'
END PROGRAM TryRetStr
```

Example: Fortran and C with a character string function.

Compile/Link/Execute

```
demo$ cc -c RetStr.c
demo$ f90 RetStr.o RetStrmain.f90
demo$ a.out
'1234*****456'
demo$ █
```


Labeled Common (f90 Calls C)

C and Fortran can share values in labeled common.

Example: A C function uses labeled common matching the Fortran one below.

```
UseCom.c
extern struct comtype {
    float p ;
    float q ;
    float r ;
} ;
extern struct comtype ilk_ ;
void usecom_ ()
{
    ilk_.p = 1.0 ;
    ilk_.q = 2.0 ;
    ilk_.r = 3.0 ;
}
```

Example: A Fortran main program uses a labeled common.

```
UseCommmain.f90
PROGRAM TryUseCom
    REAL u, v, w
    COMMON / ilk / u, v, w
    u = 7.0
    v = 8.0
    w = 9.0
    WRITE(*,*) u, v, w
    CALL UseCom ( u, v, w )
    WRITE(*,*) u, v, w
END PROGRAM TryUseCom
```

Example: Fortran and C share a labeled common.

Compile/Link/Execute

```
demo$ cc -c CUseCom.c
demo$ f90 CUseCom.o FUseCommmain.f90
demo$ a.out
    7.0000000    8.0000000    9.0000000
    1.0000000    2.0000000    3.0000000
demo$ █
```

Note – Any of the options that change size or alignment (or any equivalences that change alignment) might invalidate such sharing.

Alternate Returns (Fortran 90 Calls C) - N/A

C does not have an alternate return. The work-around is to pass an argument and branch on that.

9.5 C Calls Fortran

Arguments Passed by Reference (C Calls f90)

Simple Arguments Passed by Reference (C Calls f90)

Example: Simple arguments, Fortran arguments by reference.

SimRef.f90

```
SUBROUTINE SimRef ( b4, i4, r4, d8 )
! f90 gets passed some simple types, by reference, from C (C calls f90)
LOGICAL      b4 ! Default kind is 4-byte
INTEGER      i4 ! Default kind is 4-byte
REAL         r4 ! Default kind is 4-byte
DOUBLE PRECISION d8 ! This is 8-byte
b4 = .TRUE.
i4 = 9
r4 = 9.9
d8 = 9.9
END SUBROUTINE SimRef
```

Example: Simple arguments, C passes the *address* of each.

SimRefmain.c

```
void main ()
{ /* Simple types passed by reference to f90 (C calls f90) */
  int    b4 ; /* f90: 4-byte LOGICAL */
  int    i4 ; /* f90: 4-byte INTEGER */
  float  r4 ; /* f90: 4-byte REAL */
  double d8 ; /* f90: 8-byte DOUBLE PRECISION */
  extern simref_ ( int *, int *, float *, double * ) ;
  simref_ ( &b4, &i4, &r4, &d8 ) ;
  printf ( "%08o  %d  %3.1f %3.1f  \n",
           b4, i4, r4, d8 ) ;
}
```

Example: Simple arguments, C and Fortran.

Compile/Link/Execute

```
demo$ f90 -c SimRef.f90
demo$ cc -c SimRefmain.c
demo$ f90 SimRef.o SimRefmain.o ← This does the linking
demo$ a.out
00000001 9 9.9 9.9
demo$ █
```

Complex Arguments Passed by Reference (C Calls f90)

Example: Complex arguments, Fortran 90 expects a simple structure.

CmplxRef.f90

```
SUBROUTINE CmplxRef ( w, z )
! f90 gets passed complex arguments from C (C calls f90)
  INTEGER, PARAMETER :: doublecomplex=8
  COMPLEX w
  COMPLEX (KIND=doublecomplex) :: z
  w = ( 6, 7 )
  z = ( 8, 9 )
END SUBROUTINE CmplxRef
```

Example: Complex arguments—C passes pointers to structures.

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_ ( struct complex *, struct dcomplex * ) ;

    cmplxref_ ( w, z ) ;                      /* w and z are pointers */
    printf ( "%3.1f %3.1f \n%3.1f %3.1f \n",
             w->r, w->i, z->r, z->i ) ;
}
```

Example: Complex arguments, C and Fortran.

Compile/Link/Execute

```
demo$ f90 -c CmplxRef.f90
demo$ cc -c CmplxRefmain.c
demo$ f90 CmplxRef.o CmplxRefmain.o ← This does the linking
demo$ a.out
6.0 7.0
8.0 9.0
demo$ █
```

Character Arguments Passed by Reference (C Calls f90)

Passing strings between C and Fortran is nonstandard. It is not encouraged.

For C calling Fortran 90, if C passes a character string argument, C *must* pass the extra hidden argument because Fortran 90 expects (requires) it. For the detailed requirements of passing string arguments, see “Character Strings and Order” on page 129.

Example: Character arguments—Fortran uses extra arguments from C.

```
StrRefU.f90  SUBROUTINE StrRefU ( a, s )
              ! Character arguments -- use extra args passed from C.
              CHARACTER a*(*), s*(*)
              a = 'abcdefghi' // char(0)
              s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
              END SUBROUTINE StrRefU
```

Example: Character arguments—C passes extra arguments.

```
StrRefUmain.c  void strrefu ( char *, char *, long, long ) ; /* Declare fcn interface */
              void main ( ) /* Pass string arguments to f90 with extra args */
              {
                  char s10[10], s80[80] ; /* Provide memory for the strings */
                  long  L10, L80 ;
                  L10 = 10 ;                /* Initialize extra args */
                  L80 = 80 ;
                  strrefu_ ( s10, s80, L10, L80 ) ; /* C strings pass by reference */
                  printf ( " s10='%s' \n s80='%s' \n", s10, s80 ) ;
              }
```

Example: Character arguments—C and Fortran.

Compile/Link/Execute

```
demo$ f90 -c StrRef.f90
demo$ cc -c StrRefmain.c
demo$ f90 StrRef.o StrRefmain.o
demo$ a.out
      s10='abcdefghi'
      s80='abcdefghijklmnopqrstuvwxyz'
demo$ █
```

Vector Arguments Passed by Reference (C Calls f90)

Example: A Fortran one-dimensional array arg, implicitly indexed from 1 to 9.

```
VecRef.f90
SUBROUTINE VecRef ( v, total )
! f90 gets passed a one-dimensional array argument, from C (C calls f90 )
  INTEGER i, total, v(9)
  total = 0
  DO i = 1, 9
    total = total + v(i)
  END DO
END SUBROUTINE VecRef
```

Example: A C one-dimensional array argument, indexed from 0 to 8.

```
VecRefmain.c
void vecref_ ( int[ ], int * ) ; /* Declare fcn interface */
void main ( )
{ /* A one-dimensional array argument passed to f90 */
  int i, sum ;
  int v[9] = { 1,2,3,4,5,6,7,8,9 } ;
  vecref_ ( v, &sum ) ; /* Arrays pass by reference */
  printf ( " %d \n", sum ) ;
}
```

Example: A one-dimensional array argument, Fortran and C.

```
Compile/Link/Execute
demo$ f90 -c VecRef.f90
demo$ cc -c VecRefmain.c0
demo$ f90 VecRef.o VecRefmain.f90
demo$ a.out
45
demo$ █
```

Matrix Arguments Passed by Reference (C Calls f90)

Example: A Fortran 2 by 2 array argument, explicitly indexed from 0 to 1.

MatRef.f90

In a two-dimensional array, the rows and columns are switched, comparing C and Fortran.

```
SUBROUTINE MatRef ( a, total )
! f90 gets passed a two-dimensional array from C (C calls f90)
  INTEGER c, r, total, a(0:1,0:1)
  a(0,1) = 99                                ! Changes a(0,1)
  total = 0
  DO r = 0, 1      ! Sums all of a
    DO c = 0, 1
      total = total + a(r,c)
    END DO
  END DO
END SUBROUTINE MatRef
```

Example: A 2 by 2 C array argument, indexed from 0 to 1.

MatRefmain.c

Such square arrays are either incompatible for C and Fortran, or awkward to do right, depending on your attitude or needs.

Nonsquare arrays are worse.

```
void matref_ ( int[ ][2], int * ) ; /* Declare fcn interface */
void main ( )
{ /* A C two-dimensional array argument passed to f90 */
  int c, r, sum ;
  int m[2][2] = {{ 00, 01 },
                 { 10, 11 }} ;
  for ( c=0; c<2; c++ ) {
    for ( r=0; r<2; r++ )
      printf ( "m(%d,%d)=%#02d \n", c, r, m[c][r] ) ;
  }
  matref_ ( m, &sum ) ; /* Arrays pass by reference */
  for ( c=0; c<2; c++ ) {
    for ( r=0; r<2; r++ )
      printf ( "m(%d,%d)=%#02d \n", c, r, m[c][r] ) ;
  }
}
```

Example: A 2 by 2 array argument—show m before and after Fortran call.

Compile/Link/Execute
Before →

After →

```
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
```

← C shows that m(1,0) got changed.

Structure Arguments Passed by Reference (C Calls f90)

Example: A Fortran 90 structure argument—received from C.

StruRef.f90

See also “Complex Arguments Passed by Reference (C Calls f90)” on page 150.

```
SUBROUTINE StruRef ( n )
! f90 gets passed structure argument from C (C calls f90)
  TYPE IntReal
    INTEGER i
    REAL    r
  END TYPE IntReal
  TYPE (IntReal) n
  n % i = 8
  n % r = 9.0
END SUBROUTINE StruRef
```

Example: A C structure argument, passed to Fortran 90.

StruRefmain.c

```
struct InRl {                /* Define a structure */
    int    i ;
    float r ;
} ;

void struref_ ( struct InRl * ) ; /* Use structure, define function */

void main ( )
{
    struct InRl ir ;
    struct InRl *n = &ir ;

    n -> i = 1 ; /* Initialize the structure */
    n -> r = 2.0 ;
    struref_ ( n ) ; /* Uses Fortran routine here */
    printf ( "n->i=%d, n->r=%3.1f \n",
            n->i,    n->r ) ;
}
```

Example: A structure argument, Fortran 90 and C.

Compile/Link/Execute

```
demo$ f90 -c StruRef.f90
demo$ cc -c StruRefmain.c
demo$ f90 StruRef.o StruRefmain.o
demo$ a.out
n->i=8, n->r=9.0
demo$ █
```


Pointer Arguments Passed by Reference (C Calls f90)—N/A

These two kinds of pointers are not compatible.

Arguments Passed by Value (C Calls f90) - N/A

In general, Fortran 90 cannot pass an argument by *value*, whether Fortran 90 calls C or C calls Fortran 90.

The work-around is to pass all arguments by *reference*.

Function Return Values (C Calls f90)

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.

INTEGER Function Return Value (C Calls f90)

Example: A Fortran function with an INTEGER function return value.

```
RetInt.f90  FUNCTION RetInt ( k )  
            INTEGER k, RetInt  
            RetInt = k + 1  
            END FUNCTION RetInt
```

Example: A C program uses a Fortran function that returns an INTEGER.

```
RetIntmain.c  int retint_ ( int * ) ;    /* Declare function interface */  
  
void main()  
{  
    int k, m ;  
    k = 8 ;  
    m = 100 + retint_ ( &k ) ;  
    printf( "%d %d\n", k, m ) ;  
}
```

Example Fortran and C with an INTEGER function return value.

Compile/Link/Execute

```
demo$ f90 -c RetInt.f
demo$ cc -c RetIntmain.c
demo$ f90 RetInt.o RetIntmain.o
demo$ a.out
8 109
demo$ █
```

REAL *Function Return Value (C Calls f90)*

Example: Fortran returns a REAL to a C float.

RetFloat.f90

```
FUNCTION RetFloat ( x )
  REAL x, RetFloat
  RetFloat = x + 1.0
END FUNCTION RetFloat
```

Example: A C program uses a Fortran function that returns a REAL.

RetFloatmain.c

```
float retfloat_ ( float * ) ; /* Declare function interface */

main ( )
{
  float r, s ;
  r = 8.0 ;
  s = 100.0 + retfloat_ ( &r ) ;
  printf( " %8.6f %8.6f \n", r, s ) ;
}
```

Example Fortran and C with a REAL function return value.

Compile/Link/Execute

```
demo$ f90 -c RetFloat.f
demo$ cc -c RetFloatmain.c
demo$ f90 RetFloat.o RetFloatmain.o
demo$ a.out
8.000000 109.000000
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.

DOUBLE PRECISION *Function Return Value (C Calls f90)*

Example: Fortran function with a DOUBLE PRECISION function return value.

```
RetDbl.f90  FUNCTION RetDbl ( x )  
            DOUBLE PRECISION RetDbl, x  
            RetDbl = x + 1.0  
            END
```

Example: A C main uses a Fortran function that returns a DOUBLE PRECISION.

```
RetDblmain.c  double retdbl_ ( double * ) ; /* Declare function interface */  
  
main()  
{  
    double x, y ;  
    x = 8.0 ;  
    y = 100.0 + retdbl_ ( &x ) ;  
    printf( "%8.6f %8.6f\n", x, y ) ;  
}
```

Example Fortran and C with a DOUBLE PRECISION function return value.

Compile/Link/Execute

```
demo$ f90 -c RetDbl.f  
demo$ cc -c RetDblmain.c  
demo$ f90 RetDbl.o RetDblmain.o  
demo$ a.out  
8.000000 109.000000  
demo$ █
```

LOGICAL *Function Return Value (C Calls f90)*

Example: A Fortran function with a LOGICAL function return value.

```
RetLog.f90  FUNCTION RetLog ( b )
             LOGICAL b, RetLog
             RetLog = .NOT. b
             END FUNCTION RetLog
```

Example: A C program uses a Fortran function that returns a LOGICAL.

```
RetLogmain.c  int retlog_ ( int * ) ;    /* Declare function interface */

void main()
{
    int r, s ;
    r = 0 ;
    s = retlog_ ( &r ) ;
    printf( "%d %d\n", r, s ) ;
}
```

Example: Fortran and C with a LOGICAL function return value.

Compile/Link/Execute

```
demo$ f90 -c RetLog.f90
demo$ cc -c RetLogmain.c
demo$ f90 RetLog.o RetLogmain.o
demo$ a.out
0 1
demo$ █
```

CHARACTER *Function Return Value* (C Calls f90)

Passing strings between C and Fortran is not encouraged.

See “Character Strings and Order” on page 129, for details of compatibility.

Example: A Fortran character string function.

```
RetChr.f90  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 for C
             END FUNCTION RetChr
```

Example: A C main uses a Fortran character function.

```
RetChrmain.c  void retchr_ (char *, int , char *, int *, int) ; /* fcn interface */

main()
{ /* Use a Fortran 90 character function,(C calls f90) */
  char strbuffer[9] = "123456789" ;
  char *rval_ptr = strbuffer ;      /* extra initial arg 1 */
  int rval_len = sizeof(strbuffer) ; /* extra initial arg 2 */
  char ch = '*' ;                   /* for normal arg 1 */
  int n = 4 ;                       /* for normal arg 1 */
  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 ) ;
}
```

The function value is passed not as a “**function return value**” but as these **arguments**:

rval_ptr, pointer to string
rval_len, length of string

The normal string argument is passed as:

&ch, pointer to string
ch_len, length of string

Example: C and Fortran with a character string function.

Compile/Link/Execute

```
demo$ f90 -c RetChr.f90
demo$ cc -c RetChrmain.c
demo$ f90 RetChr.o RetChrmain.o
demo$ a.out
'123456789'
'*****'
demo$ █
```

Labeled Common (C Calls f90)

C and Fortran can share values in labeled common.

Example: A Fortran subroutine uses a labeled common.

```
UseCom.f90  SUBROUTINE UseCom
             REAL u, v, w
             COMMON / ilk / u, v, w
             u = 7.0
             v = 8.0
             w = 9.0
             END SUBROUTINE UseCom
```

Example: A C main uses a labeled common matching the Fortran one above.

```
UseCommmain.c  extern struct comtype { /* Declare a structure */
                float p ;
                float q ;
                float r ;
            } ;
                extern struct comtype ilk_ ; /* Define an item using the structure */
                void usecom_ ( ) ;          /* Declare function interface */
                void main()
                {
                    ilk_.p = 1.0 ;
                    ilk_.q = 2.0 ;
                    ilk_.r = 3.0 ;
                    usecom_ ( ) ;
                    printf(" ilk_.p=%4.1f, ilk_.q=%4.1f, ilk_.r=%4.1f\n",
                           ilk_.p, ilk_.q, ilk_.r ) ;
                }
```

Example: Fortran and C share a labeled common.

Compile/Link/Execute

```
demo$ f90 -c UseCom.f90
demo$ cc -c UseCommmain.c
demo$ f90 UseCom.o UseCommmain.o
demo$ a.out
      ilk_.p= 7.0, ilk_.q= 8.0, ilk_.r= 9.0
demo$ █
```

Note – Any of the options that change size or alignment (or any equivalences that change alignment) might invalidate such sharing.

Alternate Returns (C Calls f90)

Some C programs may need to use a Fortran routine with nonstandard returns. No new Fortran 90 routine needs alternate returns—they are *obsolete*.

Example: One regular argument and two alternate returns.

AltRet.f90

Obsolete features are candidates for removal from the next version of the standard. Fortran 90 has better ways of doing the same thing.

```
SUBROUTINE AltRet ( i, *, * ) ! Obsolete
  INTEGER i, k
  i = 9
  k = 20
  IF ( k .eq. 10 ) RETURN 1 ! Obsolete
  IF ( k .eq. 20 ) RETURN 2 ! Obsolete
  RETURN
END SUBROUTINE AltRet
```

Example: Alternate returns—C invokes the subroutine as a function.

AltRetmain.c

To C, a Fortran routine with nonstandard returns does return an int (INTEGER*4). The return value specifies which alternate return was used. If the routine has no entry points with alternate return arguments, the returned value is undefined.

```
int altret_ ( int * ) ; /* Declare function interface */
main()
{
    int k, m ;
    k = 0 ;
    m = altret_ ( &k ) ; /* Use the Fortran subroutine */
    printf( "%d %d\n", k, m ) ;
}
```

Example: Alternate returns, C and Fortran.

Compile/Link/Execute

```
demo$ f90 -c AltRet.f90
Obsolescent: The alternate RETURN construct is obsolete at line 6
Obsolescent: The alternate RETURN construct is obsolete at line 7
demo$ acc -c AltRetmain.c
demo$ f90 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 the “RETURN 2” was executed.

Features and Differences



This appendix is organized into the following sections.

<i>Standards</i>	<i>page 163</i>
<i>Extensions</i>	<i>page 164</i>
<i>Directives</i>	<i>page 179</i>
<i>Compatibility with FORTRAN 77</i>	<i>page 184</i>
<i>Forward Compatibility</i>	<i>page 188</i>
<i>Mixing Languages</i>	<i>page 188</i>
<i>Module Files</i>	<i>page 188</i>

This appendix shows some of the major features differences between:

- Standard Fortran 90 and Sun Fortran 90
- FORTRAN 77 and Fortran 90

A.1 Standards

This Fortran is an enhanced ANSI Standard Fortran 90 development system.

- It conforms to the ANSI X3.198-1992 Fortran standard and the corresponding International Standards Organization ISO/IEC 1539:1991 (E) Fortran 90 standard.
- It provides an IEEE standard 754-1985 floating-point package.

- On SPARC systems, it provides support for optimization exploiting features of SPARC V8, including the SuperSPARC™ implementation¹. These features are defined in the *SPARC Architecture Manual: Version 8*.

A.2 Extensions

Sun Fortran 90 provides the following extensions.

Tabs in the Source

£90 allows the tab character in fixed-form source and in free-form source. Standard Fortran 90 does not allow tabs.

The tab character is not converted to a blank, so the visual placement of tabbed statements depends on the utility you use to edit or display text.

Fixed-Form Source

- For a tab in column one:
 - If the next character is a nonzero digit, then the current line is a *continuation* line;
 - otherwise, the current line is an *initial* line.
- A tab cannot precede a statement label.
- A tab after column one is treated by £90 the same as a blank character, except in literal strings.

Free-Form Source

£90 treats a tab and a blank character as equivalent, except in literal strings.

1. SuperSPARC is a trademark of Texas Instruments, Inc.

Continuation Line Limits

f90 allows 99 continuation lines (1 initial and 98 continuation lines). Standard Fortran 90 allows 19 for fixed-form and 39 for free-form.

Fixed-Form Source of 96 Characters

In fixed-form source, lines can be 96 characters long. Columns 73 through 96 are ignored. Standard Fortran 90 allows 72-character lines.

Directives

f90 allows directive lines starting with `C`DIR\$, `!`DIR\$, `C`MIC\$, or `!`MIC\$. They look like comments but are not. For full details on directives, see “Directives” on page 179.

Standard Fortran 90 has no directives.

Source Form Assumed

The source form assumed by f90 depends on options, directives, and suffixes.

- Command-line options

Option	Action
<code>-fixed</code>	Interpret all source files as Fortran 90 <i>fixed</i> form
<code>-free</code>	Interpret all source files as Fortran 90 <i>free</i> form

If the `-free` or `-fixed` option is used, that overrides the file name suffix.

- File name suffixes

Suffix	Source Form
<code>.f90</code>	Fortran 90 <i>free-form</i> source files
<code>.f</code>	Fortran 90 <i>fixed-form</i> source files or ANSI standard FORTRAN 77 source files
<code>.for</code>	Same as <code>.f</code> .
<code>.ftn</code>	Same as <code>.f</code> .
<i>other</i>	None—file name is passed to the linker

- Directives

Directive	Action
<code>!DIR\$ FIXED</code>	Interpret the rest of the source file as Fortran 90 <i>fixed</i> form
<code>!DIR\$ FREE</code>	Interpret the rest of the source file as Fortran 90 <i>free</i> form

If either a `FREE` or `FIXED` directive is used, that overrides the option and file name suffix.

Mixing Forms

Some mixing of source forms is allowed.

- In the same `f90` command, some source files can be fixed form, some free.
- In the same file, free form *can* be mixed with fixed form by using directives.

Boolean Type

£90 supports constants and expressions of Boolean type. There are no Boolean variables or arrays, and there is no Boolean type statement.

Miscellaneous Rules Governing Boolean Type

- *Masking*—A bitwise logical expression has a Boolean result; each of its bits is the result of one or more logical operations on the corresponding bits of the operands.
- For binary arithmetic operators, and for relational operators:
 - If one operand is Boolean, the operation is performed with no conversion.
 - If both operands are Boolean, the operation is performed as if they were integers.
- No user-specified function can generate a Boolean result, although some (nonstandard) intrinsics can.
- Boolean and logical types differ as follows:
 - Variables, arrays, and functions can be of logical type, but they cannot be Boolean type.
 - There is a LOGICAL statement, but no BOOLEAN statement.
 - A logical variable or constant represents only one value. A Boolean constant can represent as many as 32 values.
 - A logical expression yields one value. A Boolean expression can yield as many as 32 values.
 - Logical entities are invalid in arithmetic, relational, or bitwise logical expressions. Boolean entities are valid in all three.

Alternate Forms of Boolean Constants

Fortran 90 allows a Boolean constant (octal, hexadecimal, or Hollerith) in the following alternate forms (no binary). Variables cannot be declared Boolean. Standard Fortran 90 does not allow these forms.

Octal

ddddddB, where *d* is any octal digit

- You can use the letter B or b.
- There can be 1 to 11 octal digits (0 through 7).
- 11 octal digits represent a full 32-bit word, with the leftmost digit allowed to be 0, 1, 2, or 3.
- Each octal digit specifies three bit values.
- The last (rightmost) digit specifies the content of the rightmost three bit positions (bits 29, 30, and 31).
- If less than 11 digits are present, the value is right-justified—it represents the rightmost bits of a word: bits *n* through 31. The other bits are 0.
- Blanks are ignored.

Within an I/O format specification, the letter B indicates *binary* digits; elsewhere it indicates *octal* digits.

Hexadecimal

x'ddd' or *x"ddd"*, where *d* is any hexadecimal digit

- There can be 1 to 8 hexadecimal digits (0 through 9, A-F).
- Any of the letters can be uppercase or lowercase (X, x, A-F, a-f).
- The digits must be enclosed in either apostrophes or quotes.
- Blanks are ignored.
- The hexadecimal digits may be preceded by a + or - sign.
- 8 hexadecimal digits represent a full 32-bit word and the binary equivalents correspond to the contents of each bit position in the 32-bit word.
- If less than 8 digits are present, the value is right-justified—it represents the rightmost bits of a word: bits *n* through 31. The other bits are 0.

Hollerith

<i>n</i> H...	'... 'H	"..."H
<i>n</i> L...	'... 'L	"..."L
<i>n</i> R...	'... 'R	"..."R

Above, “...” is a string of characters and *n* is the character count.

- A Hollerith constant is type Boolean.
- If any character constant is in a bitwise logical expression, the expression is evaluated as Hollerith.
- A Hollerith constant can have 1 to 4 characters.

Examples: Octal and hexadecimal constants.

Boolean Constant	Internal Octal for 32-bit word
0B	00000000000
77740B	00000077740
X"ABE"	00000005276
X"-340"	37777776300
X'1 2 3'	00000000443
X'FFFFFFFFFFFFFFFF'	37777777777

Examples: Octal and hexadecimal in assignment statements.

```
i = 1357B
j = X"28FF"
k = X'-5A'
```

Alternate Contexts of Boolean Constants

f90 allows BOZ constants in the places other than DATA statements.

B'bbb'	O'ooo'	Z'zzz'
B"bbb"	O"ooo"	Z"zzz"

If these are assigned to a real variable, no type conversion occurs.

Standard Fortran 90 allows these only in DATA statements.

Abbreviated Size Notation for Numeric Data Types

Fortran 90 allows the following nonstandard type declaration forms in declaration statements, function statements, and IMPLICIT statements.

Table A-1 Size Notation for Numeric Data Types

Nonstandard	Declarator	Short Form	Meaning
INTEGER*1	INTEGER (KIND=1)	INTEGER (1)	One-byte signed integers
INTEGER*2	INTEGER (KIND=2)	INTEGER (2)	Two-byte signed integers
INTEGER*4	INTEGER (KIND=4)	INTEGER (4)	Four-byte signed integers
LOGICAL*1	LOGICAL (KIND=1)	LOGICAL (1)	One-byte logicals
LOGICAL*2	LOGICAL (KIND=2)	LOGICAL (2)	Two-byte logicals
LOGICAL*4	LOGICAL (KIND=4)	LOGICAL (4)	Four-byte logicals
REAL*4	REAL (KIND=4)	REAL (4)	IEEE single-precision floating-point (Four-byte)
REAL*8	REAL (KIND=8)	REAL (8)	IEEE double-precision floating-point (Eight-byte)
COMPLEX*8	COMPLEX (KIND=4)	COMPLEX (4)	Single-precision complex (Four-bytes each part)
COMPLEX*16	COMPLEX (KIND=8)	COMPLEX (8)	Double-precision complex (Eight-bytes each part)

The form in column one is nonstandard. The kind number can vary by vendor.

Cray Pointers

A *Cray pointer* is a variable whose value is the address of another entity, which is called the *pointee*.

f90 supports Cray pointers. Standard Fortran 90 does not support them.

Syntax

The Cray `POINTER` statement has the following format:

```
POINTER ( pointer_name, pointee_name [array_spec] ), ...
```

Where *pointer_name*, *pointee_name*, and *array_spec* are as follows:

<i>pointer_name</i>	Pointer to the corresponding <i>pointee_name</i> . <i>pointer_name</i> contains the address of <i>pointee_name</i> . Must be: a scalar variable name (but not a structure) Cannot be: a constant, a name of a structure, an array, or a function
<i>pointee_name</i>	Pointee of the corresponding <i>pointer_name</i> Must be: a variable name, array declarator, or array name
<i>array_spec</i>	If <i>array_spec</i> is present, it must be explicit shape, (constant or nonconstant bounds), or assumed-size.

Example: Declare Cray pointers to two pointees.

```
POINTER ( p, b ), ( q, c )
```

The above example declares Cray pointer `p` and its pointee `b`, and Cray pointer `q` and its pointee `c`.

Example: Declare a Cray pointer to an array.

```
POINTER ( ix, x(n, 0:m) )
```

The above example declares Cray pointer `ix` and its pointee `x`; and declares `x` to be an array of dimensions `n` by `m-1`.

Purpose of Cray Pointers

You can use pointers to access user-managed storage by dynamically associating variables to particular locations in a block of storage.

Cray pointers allow accessing absolute memory locations.

Cray pointers do not provide convenient manipulation of linked lists because (for optimization purposes) it is assumed that no two pointers have the same value.

Cray Pointers and Fortran 90 Pointers

Cray pointers are declared as follows:

```
POINTER ( pointer_name, pointee_name [array_spec] )
```

Fortran 90 pointers are declared as follows:

```
POINTER :: object_name
```

The two kinds of pointers cannot be mixed.

Features of Cray Pointers

- Whenever the pointee is referenced, f90 uses the current value of the pointer as the address of the pointee.
- The Cray pointer type statement declares both the pointer and the pointee.
- The Cray pointer is of type Cray pointer.
- The value of a Cray pointer occupies one storage unit. Its range of values depends on the size of memory for the machine in use.
- The Cray pointer can appear in a COMMON list or as a dummy argument.
- The Cray pointee has no address until the value of the Cray pointer is defined.
- If an array is named as a pointee, it is called a *pointee array*.

Its array declarator can appear in:

- A separate type statement
- A separate DIMENSION statement
- The pointer statement itself

- If the array declarator is in a subprogram, the dimensioning can refer to:
 - Variables in a common block, or
 - Variables that are dummy arguments
- The size of each dimension is evaluated on entrance to the subprogram, not when the pointee is referenced.

Restrictions on Cray Pointers

- If *pointee_name* is of character type, it must be a variable typed `CHARACTER* (*)`.
- If *pointee_name* is an array declarator, it must be explicit shape, (constant or nonconstant bounds), or assumed-size.
- An array of Cray pointers is not allowed.
- A Cray pointer cannot be:
 - Pointed to by another Cray pointer or by a Fortran 90 pointer.
 - A component of a structure.
 - Declared to be any other data type.
- A Cray pointer cannot appear in:
 - A `PARAMETER` statement or in a type declaration statement that includes the `PARAMETER` attribute.
 - A `DATA` statement.

Restrictions on Cray Pointees

- A Cray pointee cannot appear in a `SAVE`, `DATA`, `EQUIVALENCE`, `COMMON`, or `PARAMETER` statement.
- A Cray pointee cannot be a dummy argument.
- A Cray pointee cannot be a function value.
- A Cray pointee cannot be a structure or a structure component.
- A Cray pointee cannot be of a derived type.

Note – Cray pointees can be of type character, but their Cray pointers are different from other Cray pointers. The two kinds cannot be mixed in the same expression.

Usage of Cray Pointers

Cray pointers can be assigned values as follows:

- Set to an absolute address

Example: `q = 0`

- Assigned to or from integer variables, plus or minus expressions

Example: `p = q + 100`

- Cray pointers are not integers. You cannot assign them to a real variable.
- The `LOC` function (nonstandard) can be used to define a Cray pointer.

Example: `p = LOC(x)`

Example: Use Cray pointers as described above.

```
SUBROUTINE sub ( n )
COMMON pool(100000)
INTEGER blk(128), word64
REAL a(1000), b(n), c(100000-n-1000)
POINTER ( pblk, blk ), ( ia, a ), ( ib, b ), &
        ( ic, c ), ( address, word64 )
DATA address / 64 /
pblk = 0
ia = LOC( pool )
ib = ia + 1000
ic = ib + n
...
```

Remarks about the above example:

- `word64` refers to the contents of absolute address 64
- `blk` is an array that occupies the first 128 words of memory
- `a` is an array of length 1000 located in blank common
- `b` follows `a` and is of length `n`
- `c` follows `b`
- `a`, `b`, and `c` are associated with `pool`
- `word64` is the same as `blk(17)` because Cray pointers are byte address and the integer elements of `blk` are each 4 bytes long

Optimization and Cray Pointers

For purposes of optimization, f90 assumes the storage of a pointee is never overlaid on the storage of another variable—it assumes that a pointee is not associated with another variable.

Such association could occur in either of two ways:

- A Cray pointer has two pointees, or
- Two Cray pointers are given the same value

Note – You are responsible for preventing such association.

These kinds of association are sometimes done deliberately, such as for equivalencing arrays, but then results can differ depending on whether optimization is turned on or off.

Example: `b` and `c` have the same pointer.

```
POINTER ( p, b ), ( p, c )
REAL x, b, c
p = LOC( x )
b = 1.0
c = 2.0
PRINT *, b
...
```

Above, because `b` and `c` have the same pointer, assigning 2.0 to `c` gives the same value to `b`. Therefore `b` prints out as 2.0, even though it was assigned 1.0.

Cray Character Pointers

If a pointee is declared as a character type, its Cray pointer is a Cray character pointer.

Purpose of Cray Character Pointers

A Cray character pointer is a special data type that allows f90 to maintain character strings by keeping track of the following:

- Byte address of the first character of the string
- Length
- Offset

An assignment to a Cray character pointer alters all three. That is, when you change what it points to, all three change.

Declaration of Cray Character Pointers

For a pointee that has been declared with an assumed length character type, the Cray pointer declaration statement declares the pointer to be a Cray character pointer.

- 1. Before the Cray pointer declaration statement, declare the pointee as a character type with an assumed length.**
- 2. Declare a Cray pointer to that pointee.**
- 3. Assign a value to the Cray character pointer.**

You can use functions CLOC or FCD, both nonstandard intrinsics.

Example: Declare Ccp to be a Cray character pointer and use CLOC to make it point to character string s.

```
CHARACTER*(*) a
POINTER ( Ccp, a )
CHARACTER*80  :: s = "abcdefghijklmnopqkooterwxyz"
Ccp = CLOC( s )
```

Operations on Cray Character Pointers

You can do the following operations with Cray character pointers:

```
Ccp1 + i
Ccp1 - i
i + Ccp1
Ccp1 = Ccp2
Ccp1 relational_operator Ccp2
```

where `Ccp1` and `Ccp2` are Cray character pointers and `i` is an integer.

Restrictions on Cray Character Pointers and Pointees

All restrictions to Cray pointers also apply to Cray character pointers. In addition, the following apply:

- A Cray character pointee cannot be an array.
- In a relational operation, a Cray character pointer can be mixed with only another Cray character pointer—not with a Cray pointer, not with an integer.
- A relational operation applies only to the character address and the bit offset; the length field is not involved.
- Cray character pointers must not appear in `EQUIVALENCE` statements, or any storage association statements. (The size can vary with the platform.)
- Cray character pointers are not optimized.
- Code containing Cray character pointers is not parallelized.
- A Cray character pointer in a list of an I/O statement is treated as an integer.

Intrinsics

£90 supports some intrinsic procedures which are extensions beyond the standard.

Table A-2 Nonstandard Intrinsics

Name	Definition	Type		Arguments	Arguments	Remark	Notes
		Function	Arguments				
CLOC	Get Fortran character descriptor (FCD)	Cray character pointer	character	([C=] c)			NP, I
COT	Cotangent	real	real	([X=] x)			P, E
DDIM	Positive difference	double precision	double precision	([X=] x, [Y=] y)			P, E
FCD	Create Cray character pointer in Fortran character descriptor (FCD) format	Cray pointer	i: integer or Cray pointer j: integer	([I=] i, [J=] j)		i: word address of first character j: character length	NP, I
LEADZ	Get the number of leading 0 bits	integer	Boolean, integer, real, or pointer	([I=] i)			NP, I
POPCNT	Get the number of set bits	integer	Boolean, integer, real, or pointer	([I=] i)			NP, I
POPPAR	Calculate bit population parity	integer	Boolean, integer, real, or pointer	([X=] x)			NP, I

The notes in the above table are explained as follows:

Note	Meaning
P	The name can be passed as an argument.
NP	The name cannot be passed as an argument.
E	External code for the intrinsic is called at run time.
I	£90 generates inline code for the intrinsic procedure.

A.3 Directives

A compiler *directive* directs the compiler to do some special action. Directives are also called *pragmas*.

A compiler directive is inserted into the source program as one or more lines of text. Each line looks like a comment, but has additional characters that identify it as more than a comment for this compiler. For most other compilers, it is treated as a comment, so there is some code portability.

General Directives

Currently there are only two general directives, `FREE` and `FIXED`. These directives tell the compiler to assume free-form source or fixed-form source.

Other General Directives

Some other parallel directives are included which are not described in detail because they are *not* guaranteed to be in the next release.

Table A-3 General Directives Guaranteed Only in the Current Release

Directive
<code>TASK, NOTASK</code>
<code>SUPPRESS(var1, var2, ...)</code>
<code>TASKCOMMON(cb1, cb2, ...)</code>

Form of General Directive Lines

General directives have the following syntax.

```
!DIR$ d1, d2, ...
```

A general *directive line* is defined as follows.

- A directive line starts with the 5 characters `CDIR$` or `!DIR$`, followed by:
 - A space
 - A directive
- Spaces before, after, or within a directive are ignored.
- Letters of a directive line can be in uppercase, lowercase, or mixed.

The form varies for fixed-form and free-form source as follows.

Fixed-Form Source

- Put `CDIR$` or `!DIR$` in columns 1 through 5.
- Directives are listed in columns 7 and beyond.
- Columns beyond 72 are ignored.
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.

Free-Form Source

- Put `!DIR$` followed by a space anywhere in the line.
The `!DIR$` characters are the first nonblank characters in the line (actually, non-whitespace).
- Directives are listed after the space.
- An *initial* directive line has a blank, tab, or newline in the position immediately after the `!DIR$`.
- A *continuation* directive line has a character other than a blank, tab, or newline in the position immediately after the `!DIR$`.

Thus, `!DIR$` in columns 1 through 5 works for both free-form source and fixed-form source.

FIXED *and* FREE *Directives*

These directives specify the source form of lines following the directive line.

Scope

They apply to the rest of the *file* in which they appear, or until the next FREE or FIXED directive is encountered.

Uses

- They allow you to switch source forms within a source file.
- They allow you to switch source forms for an INCLUDE file. You insert the directive at the start of the INCLUDE file. After the INCLUDE file has been processed, the source form reverts back to the form being used prior to processing the INCLUDE file.

Restrictions

The FREE/FIXED directives:

- Each must appear alone on a compiler directive line (not continued).
- Each can appear anywhere in your source code. Other directives must appear within the program unit they affect.

Example: A FREE directive.

```
!DIR$ FREE
  DO i = 1, n
    a(i) = b(i) * c(i)
  END DO
```

Parallel Directives

A *parallel* directive is a special comment that directs the compiler to do some parallelizing. Currently there is only one parallel directive, `DOALL`.

DOALL Directive

The `DOALL` directive tells the compiler to parallelize the next loop it finds, if possible.

Other Parallel Directives

Some other parallel directives are included which are not described in detail because they are *not* guaranteed to be in the next release.

Table A-4 Parallel Directives Guaranteed Only in the Current Release

Directive
CASE, END CASE
PARALLEL, END PARALLEL
DO PARALLEL, END DO
GUARD, END GUARD

Form of Parallel Directive Lines

Parallel directives have the following syntax.

```
!MIC$ DOALL [general parameters] [scheduling parameter]
```

A *parallel directive line* is defined as follows.

- A parallel directive starts with the `CMIC$` or `!MIC$`, followed by:
 - A space
 - A directive
 - For some directives, one or more parameters
- Spaces before, after, or within a directive are ignored.
- Letters of a parallel directive line can be in uppercase, lowercase, or mixed.

The form varies for fixed-form and free-form source as follows.

Fixed

- Put CMIC\$ or !MIC\$ in columns 1 through 5.
- Directives are listed in columns 7 and beyond.
- Columns beyond 72 are ignored.
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.

Free

- Put !MIC\$ followed by a space anywhere in the line.
The !MIC\$ characters are the first nonblank characters in the line (actually, non-whitespace).
- Directives are listed after the space.
- An *initial* directive line has a blank, tab, or newline in the position immediately after the !MIC\$.
- A *continuation* directive line has a character other than a blank, tab, or newline in the position immediately after the !MIC\$.

Thus, !MIC\$ in columns 1 through 5 works for both free and fixed.

Example: Directive with continuation lines (DOALL directive and parameters.)

```
!MIC$ DOALL
!MIC$&  SHARED( a, b, c, n )
!MIC$&  PRIVATE( i )
      DO i = 1, n
          a(i) = b(i) * c(i)
      END DO
```

Example: Same directive and parameters, with *no* continuation lines.

```
!MIC$ DOALL SHARED( a, b, c, n ) PRIVATE( i )
      DO i = 1, n
          a(i) = b(i) * c(i)
      END DO
```

A.4 Compatibility with FORTRAN 77

Source

Source from Sun FORTRAN 77 is not generally compatible with Fortran 90, unless it strictly follows the FORTRAN 77 standard. In general, if it uses no extensions, then it is compatible.

Executables

Libraries compiled and linked in FORTRAN 77 under Solaris 2.x run in the Fortran 90 1.0 environment.

Libraries

- Libraries (.a) and object files (.o) compiled and linked in FORTRAN 77 under Solaris 2.x are compatible with Fortran 90 1.0. You can check the /usr/4lib directory on your SunOS 5.x system for the libF77.so.2.0 and libV77.so.2.0 library files.

Example: f90 main and f77 subroutine.

```
demo$ cat m.f90
CHARACTER*74 :: c = 'This is a test.'
CALL echo1( c )
END
demo$ cat s.f
SUBROUTINE echo1( a )
CHARACTER*74 a
PRINT*, a
RETURN
END
demo$ f77 -c -silent s.f
demo$ f90 m.f90 s.o
demo$ a.out
This is a test.
demo$ ■
```

- The library `libF77` is generally compatible with `f90`.

Example: `f90` main calls a routine from the `libF77` library.

```
demo$ cat tdttime.f90
      REAL e, dttime, t(2)
      e = dttime( t )
      DO i = 1, 10000
         k = k+1
      END DO
      e = dttime( t )
      PRINT *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
      END
demo$ f90 tdttime.f90
demo$ a.out
      elapsed:6.405999884E-3, user:5.943499971E-3, sys:4.625000001E-4
demo$ █
```

See `dttime(3f)`.

I/O

`f77` and `f90` are generally I/O compatible for binary I/O, since `f90` loads the `f77` I/O compatibility library.

Such compatibility includes the following two situations:

- In the same program, you can write some records in `f90`, then read them in `f77`.
- An `f90` program can write a file. Then an `f77` program can read it.

The numbers read back in may or may not equal the numbers written out.

- Unformatted

The numbers read back in do equal the numbers written out.

- Floating-point formatted

The numbers read back in can be different from the numbers written out. This is caused by slightly different base conversion routines, or by different conventions for uppercase/lowercase, spaces, plus or minus signs, and so forth.

Examples: `1.0e12`, `1.0E12`, `1.0E+12`

- List-directed

The numbers read back in can be different from the numbers written out. This can be caused by various layout conventions with commas, spaces, zeros, repeat factors, and so forth.

Example: '0.0' as compared to '.0'

Example: ' 7' as compared to '7'

Example: '3, 4, 5' as compared to '3 4 5'

Example: '3*0' as compared to '0 0 0'

The above results are from: `integer::v(3)=(/0,0,0/); print *,v`

Example: '0.333333343' as compared to '0.333333'

The above results are from `PRINT *, 1.0/3.0`

Intrinsics

The Fortran 90 standard supports the following new intrinsic functions that FORTRAN 77 does not have.

If you use one of these names in your program, you must add an `EXTERNAL` statement to make `f90` use your function rather than the intrinsic one.

ADJUSTL	LEN_TRIM	SELECTED_INT_KIND
ADJUSTR	MAXEXPONENT	SELECTED_REAL_KIND
ALLOCATED	MINEXPONENT	SET_EXPONENT
ASSOCIATED	NEAREST	SHAPE
BIT_SIZE	PRECISION	SIZE
DIGITS	PRESENT	SPACING
EPSILON	RADIX	TINY
EXPONENT	RANGE	TRANSFER
FRACTION	REPEAT	TRIM
HUGE	RRSPACING	UBOUND
KIND	SCALE	VERIFY
LBOUND	SCAN	

The Fortran 90 standard supports the following new array intrinsic functions.

ALL	MAXLOC	RESHAPE
ANY	MAXVAL	SPREAD
COUNT	MERGE	SUM
CSHIFT	MINLOC	TRANSPOSE
DOT_PRODUCT	MINVAL	UNPACK
EOSHIFT	PACK	
MATMUL	PRODUCT	

A.5 Forward Compatibility

This next release of `f90` is intended to be source code compatible with this release.

If you generate any libraries with this release of `f90`, they are not guaranteed to be compatible with the next release.

A.6 Mixing Languages

On Solaris systems, routines written in C can be combined with Fortran 90 programs, since these languages have common calling conventions.

A.7 Module Files

If a file containing a Fortran 90 module is compiled, `f90` generates a module file (`.M` file) in addition to the `.o` file.

By default, such files are usually sought in the current working directory. The `-Mdir` option allows you to tell `f90` to seek them in an additional location.

The `.M` files cannot be stored into an archive file. If you have many `.M` files in some directory, and you want to reduce the number of such files (to reduce clutter), you can concatenate them into one large `.M` file.

iMPact: Multiple Processors



This appendix is organized into the following sections.

<i>Requirements</i>	<i>page 189</i>
<i>Overview</i>	<i>page 190</i>
<i>Speed Gained or Lost</i>	<i>page 192</i>
<i>Number of Processors</i>	<i>page 192</i>

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 parallel processing and with Sun Fortran and the SunOS or UNIX operating system.

B.1 Requirements

Multiprocessor Fortran 90 requires the following.

- SPARC multiple processor system
- Solaris 2.3 Operating Environment, or later

Solaris 2.3 and later supports the `libthread` multi-thread library and running many processors simultaneously. Fortran 90 MP has features that exploit multiple processors using Solaris 2.3 and later.

- The iMPact MT/MP multiple processor package

B.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).

Automatic Parallelization

Automatic parallelization is both fast and safe. To *automatically* parallelize loops, use the `-parallel` option. With this option, the *software* determines which loops are appropriate to parallelize.

Example: *Automatic* parallelization. Do all appropriate loops.

```
demo$ f90 -parallel any.f90
```

Explicit Parallelizing

Explicit parallelization may yield extra performance at some risk of producing 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.

Example: *Explicit* parallelization. Do only the “DO I=1, N” loop.

```
demo$ cat t1.f90
...
!MIC$ DOALL                               See Appendix D, “iMPact: Explicit Parallelization.
!MIC$&  SHARED( a, b, c, n )
!MIC$&  PRIVATE( i )
      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$ f90 -explicitpar t1.f90
```

Summary

The following table summarizes the parallel options and the directive.

Table B-1 Parallelization Summary

Options	Syntax	Risk of Incorrect Results
Explicit (<i>only</i>)	<code>-explicitpar</code>	Note directive, below.
Automatic and Explicit	<code>-parallel</code>	Note directive, below.
Automatic with reduction	<code>-parallel -reduction</code>	Note directive, below.
Directive		
DOALL	<pre>!MIC\$ DOALL !MIC\$& SHARED(v1, v2, ...) !MIC\$& PRIVATE(u1, u2, ...) other parameters</pre>	High 

Notes on the Parallel Options and the Directive

- `-parallel` includes `-explicitpar` *and* does automatic parallelization.
- The parallelization *options* can be in any order but must be all lower case.
- All require a Fortran MP enhancement package and Solaris 2.2, or later.
- 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 directive.
- You get automatic and explicit parallelization with the `-parallel` option.
 - The compiler automatically parallelizes all appropriate loops.
 - It also parallelizes any appropriate loops that you explicitly identify by a directive (still a risk with directives of producing incorrect results).
 - A loop with an explicit directive gets no reductions.

Standards

Multiprocessing is an evolving concept. When standards for multiprocessing are established, the above features may be superseded.

B.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.

$\text{speedup} = \text{vectorization} * (\text{number of CPUs} - 1)$

Remember that this is only a first-order rough approximation.

B.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` to more than `N` (usually degrades performance)
- 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 parallelizing (for overhead, other users, and so forth).
 - For a one-user system, try `PARALLEL=N-1` and try `PARALLEL=N`.
 - For a multiple-user system, 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.

iMPact: Automatic Parallelization



This appendix is organized into the following sections.

<i>What You Do</i>	<i>page 195</i>
<i>What the Compiler Does</i>	<i>page 196</i>
<i>Definition: Automatic Parallelizing</i>	<i>page 197</i>

This appendix shows an easy way to parallelize programs for multiple processors. This is called *automatic parallelizing*. This is a “how to” guide.

C.1 What You Do

See Appendix B, “iMPact: Multiple Processors” for required background.

To tell the compiler to parallelize *automatically*, use the `-parallel` option.
Example: Parallelize automatically, some loops get parallelized, some do not.

```
demo$ cat t2.f90
...
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 196, under Dependency Analysis.
END DO
...
demo$ f90 -parallel t2.f90
```

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.

C.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
REAL, TARGET :: x
REAL, POINTER :: px
EQUIVALENCE ( u, z )
s = 0.0
...
```

The variables `u`, `x`, `z`, and `px` are *scalar* variables, but *not pure scalar*.

C.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 `-parallel` option.

```

...
DO i = 1, n                      ! ← Parallelized
    a(i) = b(i) * c(i)
END DO
...
demo$ f90 -parallel t.f90

```

Exceptions for Automatic Parallelizing

For *automatic* parallelization, the compiler does not parallelize a loop if any of the following occur:

- 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.

Examples

The following examples illustrate the *definition* of what gets done with *automatic* parallelization, plus the exceptions.

Example: Using `-parallel`, 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 `-parallel`, a *constant* step size loop.

```

INTEGER, PARAMETER :: del = 2
...
DO k = 3, 1000, del           ! ← Parallelized
    x(k) = x(k) * z(k,k)
END DO
...

```

Example: Using `-parallel`, 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 `-parallel`, *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 `-parallel`, 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 `-parallel`, 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  
...
```

iMPact: Explicit Parallelization



The appendix is organized into the following sections.

See Appendix B, “iMPact: Multiple Processors for required background.

<i>What You Do</i>	<i>page 201</i>
<i>What the Compiler Does</i>	<i>page 202</i>
<i>Parallel Directives</i>	<i>page 203</i>
<i>DOALL Loops</i>	<i>page 206</i>
<i>Exceptions for Explicit Parallelizing</i>	<i>page 208</i>
<i>Risk with Explicit: Nondeterministic Results</i>	<i>page 209</i>
<i>Signals</i>	<i>page 211</i>

This appendix shows an advanced way to parallelize programs for multiple processors. This is called *explicit parallelizing*. It may be faster, with some risk of incorrect results. This is a “*how to*” guide.

D.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 `f90` command line.
- Check results very carefully.

The “!MIC\$ DOALL” is explained later.

The special directive is described later, but first it is illustrated in the following example.

Example: Parallelize the “DO I=1, N” loop *explicitly*.

```
!MIC$ DOALL
!MIC$&  SHARED( a, b, c, n )
!MIC$&  PRIVATE( i )
      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$ f90 -explicitpar t1.f90
```

D.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

D.3 Parallel Directives

Explicitly parallelizing loops requires using both of the following:

- A parallel directive
- One or more command-line options

A *parallel* directive is a special comment that directs the compiler to do some parallelizing. Directives are also called *pragmas*.

DOALL—Currently there is one parallel directive, DOALL. The compiler parallelizes the next loop it finds, if possible.

Form of Directive Lines

Parallel directives have the following syntax.

```
!MIC$ DOALL [general parameters] [scheduling parameter]
```

A *directive line* is defined as follows.

- It starts with the 5 characters CMIC\$ or !MIC\$, followed by:
 - A space
 - A directive
 - For some directives, one or more parameters
- Spaces before, after, or within a directive are ignored.
- Letters of a directive line can be in uppercase, lowercase, or mixed.

The form varies for fixed and free form source as follows.

Fixed

- Put CMIC\$ or !MIC\$ in columns 1 through 5.
- Directives are listed in columns 7 and beyond.
- Columns beyond 72 are ignored
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.

Free

- Put !MIC\$ followed by a space anywhere in the line.
The !MIC\$ characters are the first nonblank characters in the line (actually, non-whitespace).
- Directives are listed after the space.
- An *initial* directive line has a blank, tab, or newline in the position immediately after the !MIC\$.
- A *continuation* directive line has a character other than a blank, tab, or newline in the position immediately after the !MIC\$.

Thus, !MIC\$ in columns 1 through 5 works for both free and fixed.

Example: Directive with continuation lines (DOALL directive and parameters.)

```
!MIC$ DOALL
!MIC$&  SHARED( a, b, c, n )
!MIC$&  PRIVATE( i )
      DO i = 1, n
          a(i) = b(i) * c(i)
      END DO
```

Example: Same directive and parameters, with *no* continuation lines.

```
!MIC$ DOALL  SHARED( a, b, c, n )  PRIVATE( i )
      DO i = 1, n
          a(i) = b(i) * c(i)
      END DO
```

DOALL *Parameters*

The DOALL directive allows general parameters and a scheduling parameter.

Table D-1 DOALL General Parameters

Parameter	Action
IF (<i>expr</i>)	At runtime, if the expression <i>expr</i> is true, use multiprocessing. If this parameter is not specified, and the loop was not called from a parallel region, then use multiprocessing.
SHARED(<i>v1</i> , <i>v2</i> , ...)	Share the variables <i>v1</i> , <i>v2</i> , ... between parallel processes. That is, they are accessible to all the tasks.
PRIVATE(<i>x1</i> , <i>x2</i> , ...)	Do not share the variables <i>x1</i> , <i>x2</i> , ... between parallel processes. That is, each task has its own private copy of these variables.
SAVELAST	Save the values of <i>private</i> variables from the last DO iteration.
MAXCPUS(<i>n</i>)	Use no more than <i>n</i> CPUs.

Table D-2 DOALL Scheduling Parameters

Parameter	Action
SINGLE	Distribute <i>one</i> iteration to each available processor.
CHUNKSIZE(<i>n</i>)	Distribute <i>n</i> iterations to each available processor. <ul style="list-style-type: none"> <i>n</i> is an expression. For best performance, <i>n</i> must be an integer constant. Example: With 100 iterations and CHUNKSIZE(4), distribute 4 iterations to each CPU.
NUMCHUNKS(<i>m</i>)	If there are <i>i</i> iterations, then distribute <i>i/m</i> iterations to each available processor. <ul style="list-style-type: none"> There can be one smaller residual chunk. <i>m</i> is an expression. For best performance, <i>m</i> must be an integer constant. Example: With 100 iterations and NUMCHUNKS(4), distribute 25 iterations to each CPU.
GUIDED	Distribute the iterations by use of guided self-scheduling. <ul style="list-style-type: none"> This minimizes synchronization overhead, with acceptable dynamic load balancing.
VECTOR	Distribute 64 iterations to each available processor. <ul style="list-style-type: none"> If stripmining an inner loop, unrolling is used to automatically improve scheduling.

Restrictions on `DOALL` Parameters

- No one variable can be declared both shared and private.
- The loop control variable of the `DOALL` loop must be declared private.
- These variables cannot be array elements or components of derived types.
- A directive can have many *general* parameters.
- A directive can have at most one *scheduling* parameter.

D.4 `DOALL` Loops

To use explicit parallelization safely, you must understand the rules for explicit parallelizing. Explicit parallelization of a `DOALL` loop requires 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 `f90` command line.
- Insert a `DOALL` parallel directive immediately before the loop, including specifying each variable in the loop as shared or private.

Example: Explicit, DOALL loop.

```
demo$ cat t4.f90
...
!MIC$ DOALL
!MIC$&  SHARED( a, b, c, n )
!MIC$&  PRIVATE( i )
      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$ f90 -explicitpar t4.f90
```

Example: Explicit, DOALL, some calls can make dependencies.

```
demo$ cat t5.f90
...
!MIC$ DOALL
!MIC$&  SHARED( a, b, n )
!MIC$&  PRIVATE( kb, k, t )
      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$ f90 -explicitpar t5.f90
```

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.

D.5 Exceptions for Explicit Parallelizing

The following are the primary exceptions that prevent the compiler from explicitly parallelizing a `DO` loop. The compiler issues error messages that the loops are not parallelized, except for a `DO` loop nested inside another `DO` loop, which is so common that messages would be distracting.

- 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 that loop 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.

- 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.
- There is an I/O statement in the loop.

For the following exception, the compiler issues no error message.

- If you explicitly parallelize a loop, and that loop 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.
- Example: A parallelized loop with a call to a routine that also has a parallelized loop.

<pre> ... !MIC\$ DOALL !MIC\$& SHARED(a, x) !MIC\$& PRIVATE(i) DO 100 i = 1, 200 ... CALL calc (a, x) ... 100 CONTINUE ... </pre>	<pre> SUBROUTINE calc (b, y) ... !MIC\$ DOALL !MIC\$& SHARED(...) !MIC\$& PRIVATE(m) DO 1 m = 1, 1000 ... 1 CONTINUE RETURN END </pre>
--	---

↑ At runtime, loops within this subroutine do *not* run in parallel.

D.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
!MIC$ DOALL
!MIC$&  SHARED( a )
!MIC$&  PRIVATE( i )
      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$ f90 -explicitpar t1.f90

```

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.

D.7 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

!DIR\$ in directives, 180
!MIC\$ in directives, 183
.M files, 23
/usr/ccs/lib, error to specify it, 22
/usr/lib, error to specify it, 22

Numerics

132-column lines, -e, 17

A

a.out file, 10
abrupt underflow, 112
access
 named files, 42
 unnamed files, 43
accrued exceptions, do not warn, 106
actions
 actions/options sorted by action, 13
 and what options invoke them, 13
 frequently used actions/options, 13
addenda for manuals, read me file, xxi
agreement across routines, -Xlist, 69
alias, 50
 many options, short commands, 33

align
 data types, 125, 126
 double word, -dalign, 17
 errors across routines, -Xlist, 69
allocated array, 91
ANSI
 conformance check, -Xlist, 70
 X3.198-1992 Fortran standard, 163
-ansi extensions, 15
AnswerBook, documents in, xix
ar, 58
 create static library, 60
arithmetic
 nonstandard, 111
 standard, 111
array
 allocated, 91
 bounds, exceeding, 81
 C FORTRAN differences, 130
 dbx, 92, 93
 slices in dbx, 93
asa FORTRAN print, 3
audience, xvii

- automatic parallelization
 - definition, 197
 - exceptions, 198
 - overview, 190
 - usage, 195
 - what the compiler does, 196
- automatic variables, 29
- autovectorizing compiler,
 - comparison, 192

B

- Bdynamic, 15, 62
- best
 - floating point -native, 23
 - performance, 25
- binding
 - dynamic, 15, 17, 62
 - static, 15, 62
- boldface font conventions, xxii
- Boolean
 - constant, alternate forms, 168
 - type, constants, 167
- bounds of arrays
 - checking, 81
- box, clear, xxii
- BS 6832, xviii
- Bstatic, 15, 62

C

- C, 155
 - calls FORTRAN, 149
 - function return value, 141
 - is called by FORTRAN, 134
- C FORTRAN
 - function compared to subroutine, 123
 - key aspects of calls, 122
 - labeled common, 147, 160
- c, compile only, 16
- call
 - C from FORTRAN, 134
 - FORTRAN from C, 149
- CALL in a loop, parallelize, 29

- case preserving, 124
- catalog, 2
- Catalyst, 2
- catch FPE, 83, 113
- CDIR\$ in directives, 180
- cg89, 16
- cg92, 16
- CIF file, -db, 17
- clear box, xxii
- CMIC\$ in directives, 183
- code generator option, -cgyr, 16
- command
 - ar, create static library, 60
 - asa, 3
 - compiler, 9
 - f90, 9
- comments
 - as directives, 179
 - to Sun, xxi, 20
- compatibility
 - C, 188
 - FORTRAN 77, 184
 - forward, 188
 - with f77 I/O, 185
 - with f77 libraries, 185
 - with f77 object files, 184
- compile
 - check across routines, -Xlist, 71
 - fails, message, 10
 - link for a dynamic shared library, 20
 - link sequence, 10
 - link, consistent, 65
 - make assembler source files only, 30
 - only, -c, 16
 - passes, times for, 30
- compile action
 - align
 - on 8-byte boundaries, -f, 19
 - ANSI, show non-ANSI extensions,
 - ansi, 15
 - assembly-language output files, keep,
 - s, 30
 - check across routines, -Xlist, 32

-
- compile action (*continued*)
 - compile only, -c, 16
 - debug, -g, 20
 - DO loops for one trip min,
 - onetrip, 25
 - do not trap on floating-point
 - exceptions, -fnonstop, 19
 - dynamic binding
 - Bdynamic, 15, 62
 - dy, 17, 62
 - executable file, name the, -o outfil, 25
 - explicit parallelization,
 - explicitpar, 18
 - extend lines to 132 columns, -e, 17
 - fast execution, -fast, 19
 - feedback to Sun, -help, 20
 - fixed-form source, -fixed, 19
 - floating point
 - best, -native, 23
 - free-form source, -free, 19
 - generate a CIF file, -db, 17
 - generate code for
 - generic SPARC, -cg89, 16
 - SPARC, V8 -cg92, 16
 - generate double load/store
 - instructions, -dalign, 17
 - global program checking, -Xlist, 32
 - library
 - add to search path for, -Ldir, 22
 - build shared library, -G, 20
 - name a shared dynamic,
 - hname, 20
 - license
 - do not queue request,
 - noqueue, 24
 - information, -xlicinfo, 31
 - link with library x, -lx, 21
 - list of options, -help, 20
 - multi-thread safe libraries, -mt, 22
 - no automatic libraries, -nolib, 24
 - no run path, norunpath, 24
 - optimize object code, -On, 25
 - parallelize, -parallel, 26
 - pass option to other program,
 - Qoption, 27
 - paths, store into object file, -R ls, 28
 - compile action (*continued*)
 - print
 - name of each pass as compiler
 - executes, -v, 30
 - version id of each pass as
 - compiler executes, -V, 30
 - profile by
 - procedure, -p, 26
 - procedure, -pg, 27
 - reduction, analyze loops for
 - reduction, -reduction, 27
 - report execution times for
 - compilation passes,
 - time, 30
 - reset -fast so that it does not use
 - xlibmopt, 31
 - set
 - directory for temporary files,
 - temp=*dir*, 30
 - INCLUDE path, -Ipath, 21
 - module files path, -Mdir, 23
 - show commands, -dryrun, 17
 - show compile flags, -flags, 19
 - stack the local variables, -
 - stackvar, 29
 - static binding
 - Bstatic, 15, 62
 - strip executable file of symbol table, -
 - s, 29
 - use fast math routines, -
 - xlibmopt, 31
 - verbose
 - v, 30
 - compiler
 - command, 9
 - frequently used options, 13
 - passes, 30
 - recognizes files by types, 11
 - complete path name, 38
 - consistent
 - across routines, -Xlist, 69
 - arguments, commons, parameters,
 - etc., 32
 - compile and link, 11, 65
 - continuation lines, 165

conventions in text, xxii
Courier font, xxii
Cray
 character pointer, 176
 pointer, 171
 pointer and Fortran 90 pointer, 172
 pointer and optimization, 175
create
 library, 59
 SCCS files, 53
cross reference table, -Xlist, 32, 76
current working directory, 37

D

-dalign, double-word align, 17
data
 inspection, dbx, 96
-db CIF file, 17
dbx, 77
 allocated arrays, 91
 arrays, 92
 catch FPE, 82, 83
 commands, 96
 current procedure and file, 96
 debug, 3
 f90 -g, 20
 -g, 79
 locate exception
 by line number, 82, 83
 next, 81
 print, 80
 quit, 79
 run, 80
 set breakpoint, 79
 structures, 85, 86, 87, 88, 89
debug, 113

allocated arrays, 91
arguments, agree in number and
 type, 69
array slices, 93
arrays, 92
checking across routines for global
 consistency, 69
column print, 93
common blocks, agree in size and
 type, 69
dbx, 3
debugger, 3
generic function, 94
IEEE exceptions, 113
locating exception
 by line number, 83
option, -g, 20
parameters, agree globally, 69
pointer, 90
 to a scalar, 85
 to an array, 86
 to user defined type, 89
record, 90
row print, 93
slices of arrays, 93
stack trace, 84
structure, 85, 86, 87, 88, 89
trace of calls, 84
uppercase, lowercase, 96
user defined type, 87
debugger, main features, 96
declared but unused, checking,
 -Xlist, 70
deep, vasty, 119
dependency
 analysis, 196
 with explicit parallelization, 210
diamond indicates nonstandard, xxii
differences
 Fortran 90, standard, Sun, FORTRAN
 77, 163
direct I/O, 45

-
- directive, 165, 179, 182, 203
 - DOALL, 182
 - explicit parallelization, 180, 182, 203
 - line defined, 180
 - directory, 37
 - current working, 37
 - object library search, 22
 - temporary files, 30
 - display to terminal, -Xlist, 71
 - division by zero IEEE, 101
 - dn, 17, 62
 - DO loops executed once, -onetrip, 25
 - DOALL directive, 182
 - doall loop, 206
 - double-word align, -dalign, 17
 - dryrun, 17
 - dy, 17, 62
 - dynamic
 - binding, 17, 62
 - library, 61
 - build, -G, 20
 - name a dynamic library, 20
 - path in executables, 28
- E**
- e, extended source lines, 17
 - ed, 2
 - emacs, 2
 - email
 - alias, Sun Programmers SIG, 227
 - send feedback comments to Sun, xxi
 - environment
 - variable, shorten command line, 33
 - EOS package, 2
 - era, 2
 - errata and addenda for manuals, read me file, xxi
 - error
 - standard error, 41, 44
 - standard error, accrued exceptions, 111
 - errors only, -XlistE, 75
 - establish a signal handler, 109
 - event management, dbx, 96
 - ex, 2
 - exceptions
 - debugging, 113
 - explicit parallelization, 208
 - handlers, 102, 107
 - ieee_handler, 107
 - location in dbx
 - by line number, 83
 - unrequired, 111
 - executable file
 - built-in path to dynamic libraries, 28
 - names in, nm command, 60
 - naming it, 25
 - strip symbol table from, -s, 29
 - execution time
 - compilation passes, 30
 - optimization, 25
 - explicit
 - parallelization, 201
 - exceptions, 208
 - overview, 190
 - risk, 209
 - explicitpar, parallelize explicitly, 18
 - extended
 - lines, -e, 17
 - syntax check, -Xlist, 70
 - extensions
 - non-ANSI, 15
 - to Fortran 90, 164
- F**
- f, align on 8-byte boundaries, 19
 - f90 command, 9
 - fast, fast execution, 19
 - features
 - debugger, 96
 - Fortran 90, standard, Sun, FORTRAN 77, 163
 - feedback file for email to Sun, xxi
 - feedback to Sun, -help, 20
 - FFLAGS shorten command line, 33

file
 a.out, 10
 directory, 37
 executable, 10
 information files, xxi
 internal, 46
 object, 10
 permissions C FORTRAN, 133
 pipe, 41
 redirection, 40
 split by `fsplit`, 3
 standard error, 44
 standard input, 44
 standard output, 44
 system, 35
file names, 42
 passing to programs, 43
 recognized by the compiler, 11, 165, 166
FIPS 69-1, xviii
fixed
 form source, 180
 form source and tabs, 164
FIXED directive, 179
 -`fixed` form source, 19
 -`flags` synonym for `-help`, 19
floating-point
 exceptions, `-fnonstop`, 19
 Goldberg paper, xix
 hardware, 33
 option, `-native`, 23
 -`fnonstop` no stop on floating-point exceptions, 19
font
 boldface, xxii
 conventions, xxii
 Courier, xxii
 italic, xxii
FORTRAN
 calls C, 134
 is called by C, 149
 read me file, bugs, new/changed features, xxi
Fortran print, `fpr`, 3
FPE catch in dbx, 83
fpr FORTRAN print, 3
fpversion, show floating-point version, 33
free
 form source, 180
 form source and tabs, 164
FREE directive, 179, 181
 -`free`, free-form source, 19
fsplit FORTRAN file split, 3
function
 called within a loop,
 parallelization, 208
 compared to subroutine, C
 FORTRAN, 123
 data type of, checking, `-xlist`, 70
 names, 124
 return values from C, 141
 return values to C, 155
 unused, checking, `-xlist`, 70
 used as a subroutine, checking,
 `-xlist`, 70

G

 -`g`, debug, 20
 -`G`, generate a dynamic library, 20
generic functions, debug, 94
getcwd, 37
Glendower, 119
global program checking, 69
Goldberg, floating-point white paper, xix
gprof
 -`pg`, profile by procedure, 27
gradual underflow, 112
graphically monitor variables, dbx, 97
GSA validated, xviii
guidelines for number of processors, 193

H

- h name, 20
- handlers, exception, 102, 107
- hardware
 - floating-point fpversion, 33
- help, 20
- help, list of options, 20
- Henry IV, 119
- hexadecimal, 168
- hierarchical file system, 35
- Hollerith, 169
- Hotspur, 119

I

- I/O, 40
- identifiers and lowercase, 124
- IEEE, 101, 111, 113
 - 754, xviii
 - exceptions, 102
 - signal handler, 109
 - standard 754-1985, 163
 - warning messages off, 106
- ieee_flags, 104, 105
- ieee_functions, 104
- ieee_handler, 104, 107
- ieee_values, 104, 106
- impatient user's guide, 5
- INCLUDE path, 21
- inconsistency
 - arguments, checking, -Xlist, 70
 - named common blocks, checking, -Xlist, 70
- increase stack size, 30
- indeterminacy, how it arises, 210
- index check of arrays, 81
- information files, xxi
- input
 - redirection, 40
 - standard, 44
- inserting SCCS ID keywords, 53
- installation directory, 65

interface

- for C and FORTRAN, 119
- problems, checking for, -Xlist, 70

internal files, 46

intrinsic procedures, extensions, 178

invalid, IEEE exception, 101

-Ipath, INCLUDE files, 21

italic font conventions, xxii

L

labeled common C FORTRAN, 147, 160

labels, unused, -Xlist, 70

LD_LIBRARY_PATH, 66

LD_RUN_PATH, 66

- and -R, not identical, 28

-Ldir, 22

libm, user error making it unavailable, 22

libraries

- C FORTRAN, 131
- paths in executables, 28
- search order, 65

library, 57

- build, -G, 20
- create, 59
- load, 21
- loaded, 58
- name a shared library, 20
- paths in executables, 28
- replace module, 61
- static, 58

license

- information, 31
- no queue, 24

licensing, 3

limit stack size, 29

line length, 165

line number of

- exception, 83
- segmentation fault (SIGSEGV), 81

line-numbered listing, -Xlist, 71

lines extended -e, 17

- link
 - options, 65
 - sequence, 10
 - suppress, 16
- linker, search order, 65
- lint-like checking across routines,
 - Xlist, 69
- list of options, -help, 20
- listing
 - line numbered with diagnostics,
 - Xlist, 69
 - Xlist, 76
- load
 - library, 21
 - map, 58
- loaded library, 58
- local
 - variables, 29
- locating
 - exception
 - by line number, 83
 - segmentation fault by line
 - number, 82
- long command lines, 33
- loop
 - parallelizing a CALL in a loop, 29
- lowercase, do not convert to, 124

M

- M files, .M files, 23
- m linker option for load map, 58
- main stack, 29
- make, 50
- making SCCS directory, 52
- man pages, xix
- manuals, xix
- many options, short commands, 33
- map, load, 58
- mateo, 84
- math
 - library, user error making it
 - unavailable, 22

- mdir modules directory, 23
- membership in SunPro SIG, Sun
 - Programmers Special Interest
 - Group, 227
- MIL-STD-1753, xviii
- miscellaneous tips
 - alias, many options, short
 - commands, 33
 - environment variable, many options,
 - short commands, 33
 - floating-point version, 33
- mixing form of source lines, 166
- monitor variables graphically, dbx, 97
- MP FORTRAN, 189
- mt, multi-thread safe libraries, 22
- multiprocessing standards, 191
- multiprocessor FORTRAN, 189

N

- name
 - compiler pass, show each, 30
 - executable file, 25
- names in executable, nm command, 60
- native floating point, 23
- NBS validated, xviii
- nesting
 - parallelized loops, 198, 208
- network licensing, 3
- NIST validated, xviii
- nm, names in executable, 60
- no license queue, 24
- nolib, 24
- non-ANSI extensions, 15
- nondeterministic results, explicit
 - parallelization, 209
- nonstandard
 - arithmetic, 111
 - indicated by diamond, xxii
- noqueue, 24
- norunpath, 24

number of
processors for parallelization, 192

O

-O, 25
with -g, 25
-o, output file, 25
-O1, 25, 26
-O2, 25
-O3, 25
object library search directories, 22
obscurities, checking for -Xlist, 70
octal, 168
ode to trace, 84
off
license queue, 24
link system libraries, 24
linking, 16
trap for floating-point exceptions, 19
warnings
IEEE accrued exceptions, 106
-xlibmopt, 31
-onetrip, 25
on-line
documents, xviii, xix
optimization
object code, 25
performance, 19
options, 12
and what actions they do, 15
frequently used, 13
list available options, -help, 20
listed by
option name, 15
what they do, 13
most frequently used, 12
options/actions sorted by option, 15
show list of, -help, 20
OPTIONS variable for command line, 33
order of linker search, 66

output
file, naming it, 25
redirection, 40
standard, 44
to terminal, -Xlist, 71

overflow
IEEE, 101
stack, 29

P

-p, profile by procedure, 26
parallel directive, 182
PARALLEL, number of processors, 193
-parallel, parallelize loops, 26
parallelization
automatic, 195
CALL in a loop, 29
explicit, 18, 201
general requirements, 189
number of processors, 193
overview, 190
reduction, 27
speed gained or lost, 192
summary table, 191
part numbers for manuals, xix
parts of large arrays in dbx, 93
pass
arguments by value, 127
file names to programs, 43
passes of the compiler, 30
path, 36
.M files, 23
built in during build of a.out, 67
INCLUDE files, 21
modules files, 23
path name, 38
absolute, 38
complete, 38
relative, 38
performance
optimization, 19
-pg, profile by procedure, 27
pipes, 41

pointee, 171
 pointer, 171
 pointer, debug, 90
 porting
 problems, checking, -Xlist, 70
 position-independent code, 61
 pragma, 179, 203
 preconnected units, 44
 prerequisites, xvii
 preserve case, 124
 print
 array
 parts of large, in dbx, 93
 slices in dbx, 93
 asa, 3
 fpr, 3
 PRIVATE parameter of DOALL, 183
 procedure
 profile -pg gprof, 27
 process control, dbx, 96
 processors
 number for parallelization, 192
 prof, -p, 26
 profile by
 procedure, -p, prof, 26
 procedure, -pg, gprof, 27
 prompt
 conventions, xxii
 pure scalar variable, 197
 purpose of manual, xvii
 pwd, 37

Q

-Qoption, 27

R

-R and LD_RUN_PATH, not identical, 28
 -R list, store lib paths, 28
 -r option for ar, 61
 random I/O, 45
 READMEs directory, xxi

record debug, 90
 recursive I/O, 23
 redirection, 40
 standard error, 41
 -reduction, parallelize automatically,
 with reduction, 27
 reference
 versus value, C/FORTRAN, 127
 referenced but not declared, checking,
 -Xlist, 70
 relative path name, 38
 rename executable file, 6
 replace library module, 61
 retrospective of accrued exceptions, 111
 return function values to C, 155
 risk with explicit parallelization, 209
 root, 36
 run path in executable, 24
 running FORTRAN, 6

S

-S, 30
 -s, 29
 safe libraries for multi-thread
 programming, 22
 sample interface C FORTRAN, 120
 SCCS, 52
 checking in files, 54
 checking out files, 54
 creating files, 53
 inserting keywords, 53
 making directory, 52
 putting files under SCCS, 52
 search
 object library directories, 22
 segmentation fault, 29, 82
 some causes, 81

set
 directory for
 temporary files, 30
 INCLUDE path, 21
 number of
 processors for
 parallelization, 192
 Shakespeare, 119
 shared library
 name a shared library, 20
 SHARED parameter of DOALL, 183
 shell script, 49
 shorten command lines
 alias, 33
 environment variable, 33
 show commands, 17
 SIG, Sun Programmers Special Interest Group, xxi, 227
 SIGFPE
 debugging, 113
 definition, 102, 107
 detect in dbx, 113
 generate, 107
 when generated, 109, 113
 signal
 handler, 109
 with explicit parallelization, 211
 SIGSEGV, some causes, 81
 size
 of data types, 125, 126
 slices of arrays in dbx, 93
 Solaris, 2
 source
 lines -e, 17
 source form
 directives, 166
 options, 165
 suffixes, 166
 speed gained or lost from
 parallelization, 192
 spirits, 119
 stack
 overflow, 29
 variables, 29
 stack trace, 84
 -stackvar, 29
 standard
 arithmetic, 112
 conformance to standards, xviii
 error, 41
 error, accrued exceptions, 111
 Fortran 90, 163
 input, 40, 44
 output, 40, 44
 statement
 unreachable, checking, -xlist, 70
 static
 binding, 17, 62
 library, 58
 strip executable of symbol table, -s, 29
 structure
 debug, 87, 88, 89
 stupid UNIX tricks
 shorten command line, alias, 33
 shorten command line, environment variable, 33
 subprogram in loop, explicit
 parallelization, 208
 subroutine
 compared to function, C
 FORTRAN, 123
 unused, checking, -xlist, 70
 used as a function, checking,
 -xlist, 70
 suffix
 of file names recognized by
 compiler, 11, 165, 166
 Sun Programmer Quarterly
 Newsletter, 227
 Sun, sending feedback to, xxi, 20
 SunOS
 5.x, 2

suppress
 error nnn, -Xlist, 75
 license queue, 24
 linking, 16
 trap for floating-point exceptions, 19
 warnings
 -Xlist, 76

SVR4, 2

symbol table
 for dbx, -g, 20
 strip executable of, 29

syntax
 compiler, 9
 errors, -Xlist, 70
 f90, 9

System V Release 4 (SVR4), 2

T

tab character in source, 164

-temp, 30

templates inline, 21

temporary files, directory for, 30

terminal display, -Xlist, 71, 76

textedit, 2

third-party software and hardware, 2

thread stack, 29

-time, timing compilation passes, 30

traceback
 dbx, 84
 ode, 84

tree, 36

triangle as blank space, xxii

turn off warnings about IEEE accrued exceptions, 106

type checking across routines, -Xlist, 70

type declaration alternate form, 170

typewriter font, xxii

U

-U do not convert to lowercase, 124

underflow
 abrupt, 112
 gradual, 112
 IEEE, 101

units, preconnected, 44

unrecognized options, 12

unrequited exceptions, 111

unused functions, subroutines, variables,
 labels, -Xlist, 70

uppercase
 debug, 96
 external names, 124

usage
 automatic parallelization, 195
 compiler, 9
 explicit parallelization, 201

V

-V, 30

-v, 30

variable
 unused, checking, -Xlist, 70
 used but unset, checking, -Xlist, 70

vasty deep, 119

verify agreement across routines,
 -Xlist, 69

version
 id of each compiler pass, 30

vi, 2

W

-w, 31

watchpoints, dbx, 97

where
 exception occurred, by line
 number, 83
 execution stopped, 84

wimp, 77

 interface to dbx, 3, 96

X

- xemacs, 2
- xlibmopt, use fast math routines, 31
- xlicinfo, 31
- Xlist, 71
 - a la carte options, 73
 - combination special, 73
 - defaults, 71
 - display directly to terminal, 71
 - errors and
 - cross reference, -XlistX, 74
 - listing, -XlistL, 74
 - sample usage, 72
 - suboptions, 73
 - details, 75
 - summary, 32, 74
- Xlist, global program checking, 32, 69
- XlistE, 74, 75
- Xlisterr, 75
- XlistI, 75
- Xlistln, 76
- Xlisto, 76
- Xlistw, 76
- Xlistwar, 76
- XlistX, 76
- xnolib, 31, 32
- xnolibmopt, 31
- xOn, 32
- xpg, 32

Z

- zero, division by, 101

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
You receive one free CD-ROM for joining, plus 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 SUN 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	I'd like to pay for my one-year membership fee of \$20 by: <input type="checkbox"/> VISA <input type="checkbox"/> MASTERCARD Card # _____ Expiration Date: _____ Signature: _____ <input type="checkbox"/> Check made payable to SunSoft	
Name		
Title		
Company		
Email Address		
Address		
City		State
ZIP		Country
Phone		
Fax		
ALL INFO MUST BE FILLED OUT SunSoft, A Sun Microsystems, Inc. Business		