Fortran Programming Guide

Sun WorkShop 6
Fortran 95
Fortran 77
Important Note on New Product Names

As part of Sun’s new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun’s core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

<table>
<thead>
<tr>
<th>Old Product Name</th>
<th>New Product Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun Visual WorkShop™ C++</td>
<td>Forte™ C++ Enterprise Edition 6</td>
</tr>
<tr>
<td>Sun Performance WorkShop™ Fortran</td>
<td>Forte™ for High Performance Computing 6</td>
</tr>
<tr>
<td>Sun Performance WorkShop™ Fortran Personal Edition</td>
<td>Forte™ Fortran Desktop Edition 6</td>
</tr>
<tr>
<td>Sun WorkShop Professional™ C</td>
<td>Forte™ C 6</td>
</tr>
<tr>
<td>Sun WorkShop™ University Edition</td>
<td>Forte™ Developer University Edition 6</td>
</tr>
</tbody>
</table>

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.

- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.
Contents

Preface 1

1. Introduction 11
   Standards Conformance 11
   Features of the Fortran Compilers 12
   Other Fortran Utilities 13
   Debugging Utilities 13
   Sun Performance Library 14
   Interval Arithmetic 14
   Man Pages 14
   READMEs 16
   Command-Line Help 17

2. Fortran Input/Output 19
   Accessing Files From Within Fortran Programs 19
      Accessing Named Files 19
      Opening Files Without a Name 21
      Preconnected Units 21
      Opening Files Without an OPEN Statement 22
   Passing File Names to Programs 22
Special Compiler Options 77
  Subscript Bounds (-c) 77
  Undeclared Variable Types (-u) 77
  Version Checking (-v) 78

Interactive Debugging With dbx and Sun WorkShop 78
f77: Viewing Compiler Listing Diagnostics 79

6. Floating-Point Arithmetic 81
  Introduction 81
  IEEE Floating-Point Arithmetic 82
    -ftrap=mode Compiler Options 83
    Floating-Point Exceptions and Fortran 83
    Handling Exceptions 84
    Trapping a Floating-Point Exception 84
    SPARC: Nonstandard Arithmetic 84
  IEEE Routines 85
    Flags and ieee_flags() 86
    IEEE Extreme Value Functions 90
    Exception Handlers and ieee_handler() 91
    Retrospective Summary 97
  Debugging IEEE Exceptions 97
  Further Numerical Adventures 100
    Avoiding Simple Underflow 100
    Continuing With the Wrong Answer 101
    SPARC: Excessive Underflow 102
    Interval Arithmetic 103

7. Porting 105
  Time and Date Functions 105
Contents

Formats 109
Carriage-Control 109
Working With Files 110
Porting From Scientific Mainframes 110
Data Representation 111
Hollerith Data 111
Nonstandard Coding Practices 114
  Uninitialized Variables 115
  Aliasing Across Calls 115
  Obscure Optimizations 115
Troubleshooting 118
  Results Are Close, but Not Close Enough 118
  Program Fails Without Warning 119

8. Performance Profiling 121
  Sun WorkShop Performance Analyzer 121
  The time Command 122
    Multiprocessor Interpretation of time Output 122
  The gprof Profiling Command 123
    Overhead and Other Considerations 127
  The tcov Profiling Command 127
    “Old Style” tcov Coverage Analysis 128
    “New Style” Enhanced tcov Analysis 130
  f77 I/O Profiling 131

9. Performance and Optimization 135
  Choice of Compiler Options 135
    Performance Option Reference 136
  Other Performance Strategies 141
10. SPARC: Parallelization 145

Essential Concepts 145
Speedups—What to Expect 146
Steps to Parallelizing a Program 147
Data Dependency Issues 148
Parallel Options and Directives Summary 149
Number of Threads 151
Stacks, Stack Sizes, and Parallelization 152

Automatic Parallelization 153
Loop Parallelization 153
Arrays, Scalars, and Pure Scalars 154
Automatic Parallelization Criteria 154
Automatic Parallelization With Reduction Operations 156

Explicit Parallelization 159
Parallelizable Loops 160
Sun-Style Parallelization Directives 165
Cray-Style Parallelization Directives 176

Environment Variables 178
PARALLEL and OMP_NUM_THREADS 179
SUNW_MP_THR_IDLE 179

Debugging Parallelized Programs 179
First Steps at Debugging 179
Debugging Parallel Code With dbx 181
11. C-Fortran Interface 183

Compatibility Issues 183
  Function or Subroutine? 184
  Data Type Compatibility 184
  Case Sensitivity 186
Underscores in Routine Names 187
Argument-Passing by Reference or Value 188
Argument Order 188
Array Indexing and Order 188
File Descriptors and stdio 189
File Permissions 190
Libraries and Linking With the $f77$ or $f95$ Command 191
Fortran Initialization Routines 191
Passing Data Arguments by Reference 192
  Simple Data Types 192
  COMPLEX Data 193
  Character Strings 193
One-Dimensional Arrays 194
Two-Dimensional Arrays 195
Structures 196
Pointers 198
Passing Data Arguments by Value 198
Functions That Return a Value 199
  Returning a Simple Data Type 200
  Returning COMPLEX Data 200
  Returning a CHARACTER String 201
Labeled COMMON 203
Sharing I/O Between Fortran and C  203
Alternate Returns  204

Index  205
Tables

TABLE 1-1 READMEs of Interest 16
TABLE 2-1 csh/sh/ksh Redirection and Piping on the Command Line 26
TABLE 4-1 Major Libraries Provided With the Compilers 60
TABLE 5-1 Xlist Suboptions 70
TABLE 5-2 Summary of Xlist Suboptions 71
TABLE 6-1 ieee_flags( action, mode, in, out ) Argument Values 87
TABLE 6-2 ieee_flags Argument Meanings 87
TABLE 6-3 Functions Returning IEEE Values 90
TABLE 6-4 Arguments for ieee_handler( action, exception, handler ) 92
TABLE 7-1 Fortran Time Functions 106
TABLE 7-2 Summary: Nonstandard VMS Fortran System Routines 107
TABLE 7-3 f77: Maximum Characters in Data Types 112
TABLE 9-1 Some Effective Performance Options 136
TABLE 10-1 Parallelization Options 149
TABLE 10-2 Sun-Style Parallel Directives 150
TABLE 10-3 Recognized Reduction Operations 157
TABLE 10-4 Explicit Parallelization Problems 162
TABLE 10-5 DOALL Qualifiers 168
TABLE 10-6 DOALL SCHEDTYPE Qualifiers 172
TABLE 10-7  DOALL Qualifiers (Cray Style)  177
TABLE 10-8  DOALL Cray Scheduling  178
TABLE 11-1  Data Sizes and Alignments—(in Bytes) Pass by Reference  (f77 and cc)  185
TABLE 11-2  Data Sizes and Alignment—(in Bytes) Pass by Reference  (f95 and cc)  186
TABLE 11-3  Comparing I/O Between Fortran and C  190
TABLE 11-4  Passing Simple Data Types  192
TABLE 11-5  Passing COMPLEX Data Types  193
TABLE 11-6  Passing a CHARACTER string  194
TABLE 11-7  Passing a One-Dimensional Array  194
TABLE 11-8  Passing a Two-Dimensional Array  195
TABLE 11-9  Passing FORTRAN 77 STRUCTURE Records  196
TABLE 11-10 Passing Fortran 95 Derived Types  197
TABLE 11-11 Passing a FORTRAN 77 POINTER  198
TABLE 11-12 Passing Simple Data Arguments by Value: FORTRAN 77 Calling C  199
TABLE 11-13 Functions Returning a REAL or float Value  200
TABLE 11-14 Function Returning COMPLEX Data  201
TABLE 11-15 A Function Returning a CHARACTER String  202
TABLE 11-16 Emulating Labeled COMMON  203
TABLE 11-17 Alternate Returns  204
Preface

This guide presents the essential information programmers need to develop efficient applications using the Sun WorkShop™ Fortran compilers, $f77$ (Fortran 77) and $f95$ (Fortran 95). It presents issues relating to input/output, program development, use and creation of software libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and the C/Fortran interface.

Discussion of the compiler command-line options and their use can be found in the companion book, the *Fortran User’s Guide*.

This guide is intended for scientists, engineers, and programmers who have a working knowledge of the Fortran language and wish to learn how to use the Sun Fortran compilers effectively. Familiarity with the Solaris operating environment or UNIX® in general is also assumed.

Multiplatform Release

This Sun WorkShop Fortran release supports versions 2.6, 7, and 8 of the Solaris™ SPARC™ Platform Edition Operating Environment.

See the README files *fortran_77* and *fortran_95* in the Sun WorkShop READMEs directory for information regarding availability of this release of the $f77$ and $f95$ compilers on specific platforms. (See page 16.)
Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

1. Display the current value of the `PATH` variable by typing:

   ```bash
   % echo $PATH
   ```

2. Review the output for a string of paths containing `/opt/SUNWspro/bin/`.

   If you find the paths, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your `PATH` environment variable by following the instructions in this section.

To determine if you need to set your `MANPATH` environment variable:

1. Request the `workshop` man page by typing:

   ```bash
   % man workshop
   ```

2. Review the output, if any.

   If the `workshop(1)` man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your `MANPATH` environment variable.

**Note** – The information in this section assumes that your Sun WorkShop 6 products were installed in the `/opt` directory. Contact your system administrator if your Sun WorkShop software is not installed in `/opt`.

The `PATH` and `MANPATH` variables should be set in your home `.cshrc` file if you are using the C shell or in your home `.profile` file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your `PATH` variable:

  ```bash
  /opt/SUNWspro/bin
  ```
To access Sun WorkShop man pages with the man command, add the following to your MANPATH variable:

    /opt/SUNWspro/man

For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this release, see the Sun WorkShop 6 Installation Guide or your system administrator.

How This Book Is Organized

Chapter 1, “Introduction,” briefly describes the features of the compilers.

Chapter 2, “Fortran Input/Output,” discusses how to use I/O efficiently.

Chapter 3, “Program Development,” demonstrates how program management tools like SCCS, make, and Teamware can be helpful.

Chapter 4, “Libraries,” explains use and creation of software libraries.

Chapter 5, “Program Analysis and Debugging,” describes use of dbx and other analysis tools.

Chapter 6, “Floating-Point Arithmetic,” introduces important issues regarding numerical computation accuracy.

Chapter 7, “Porting,” considers porting programs to Sun compilers.

Chapter 8, “Performance Profiling,” describes techniques for performance measurement.

Chapter 9, “Performance and Optimization,” indicates ways to improve execution performance of Fortran programs.

Chapter 10, “SPARC: Parallelization,” explains the multiprocessing features of the compilers.

Chapter 11, “C-Fortran Interface,” describes how C and Fortran routines can call each other and pass data.
Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1  Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. % You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td>% su Password:</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized</td>
<td>Read Chapter 6 in the User's Guide. These are called class options. You must be superuser to do this.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Command-line placeholder text; replace with a real name or value</td>
<td>To delete a file, type rm filename.</td>
</tr>
</tbody>
</table>

- The symbol Δ stands for a blank space where a blank is significant:

\[
\Delta 36.001
\]

- FORTRAN 77 examples appear in tab format, while Fortran 95 examples appear in free format. Examples common to both Fortran 77 and 95 use tab format except where indicated.

- The FORTRAN 77 standard uses an older convention of spelling the name "FORTRAN" capitalized. Sun documentation uses both FORTRAN and Fortran. The current convention is to use lower case: "Fortran 95".

- References to online man pages appear with the topic name and section number. For example, a reference to GETENV will appear as getenv(3F), implying that the man command to access this page would be: man -s 3F getenv

- System Administrators may install the Sun WorkShop Fortran compilers and supporting material at: <install_point>/SUNWspro/ where <install_point> is usually /opt for a standard install. This is the location assumed in this book.
Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell</td>
<td>%</td>
</tr>
<tr>
<td>Bourne shell and Korn shell</td>
<td>$</td>
</tr>
<tr>
<td>C shell, Bourne shell, and Korn shell superuser</td>
<td>#</td>
</tr>
</tbody>
</table>

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the docs.sun.com<sup>sm</sup> Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:
  
  http://docs.sun.com

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:

  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:
    
    /opt/SUNWspro/docs/index.html

  (Contact your system administrator if your Sun WorkShop software is not installed in the /opt directory.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document’s title.
TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

**TABLE P-3**  
Related Sun WorkShop 6 Documentation by Document Collection

<table>
<thead>
<tr>
<th>Document Collection</th>
<th>Document Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forte™ Developer 6 / Sun WorkShop 6 Release Documents</td>
<td>About Sun WorkShop 6 Documentation</td>
<td>Describes the documentation available with this Sun WorkShop release and how to access it.</td>
</tr>
<tr>
<td></td>
<td>What’s New in Sun WorkShop 6</td>
<td>Provides information about the new features in the current and previous release of Sun WorkShop.</td>
</tr>
<tr>
<td></td>
<td>Sun WorkShop 6 Release Notes</td>
<td>Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.</td>
</tr>
<tr>
<td>Forte Developer 6 / Sun WorkShop 6</td>
<td>Analyzing Program Performance With Sun WorkShop 6</td>
<td>Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool er_print, the LoopTool and LoopReport utilities, and UNIX profiling tools prof, gprof, and tcov.</td>
</tr>
<tr>
<td></td>
<td>Debugging a Program With dbx</td>
<td>Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.</td>
</tr>
<tr>
<td></td>
<td>Introduction to Sun WorkShop</td>
<td>Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.</td>
</tr>
<tr>
<td>Document Collection</td>
<td>Document Title</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Forte™ C 6 / Sun WorkShop 6 Compilers C</td>
<td>C User’s Guide</td>
<td>Describes the C compiler options, Sun-specific capabilities such as pragmas, the lint tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.</td>
</tr>
<tr>
<td>Forte™ C++ 6 / Sun WorkShop 6 Compilers C++</td>
<td>C++ Library Reference</td>
<td>Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.</td>
</tr>
<tr>
<td></td>
<td>C++ Migration Guide</td>
<td>Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.</td>
</tr>
<tr>
<td></td>
<td>C++ Programming Guide</td>
<td>Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.</td>
</tr>
<tr>
<td></td>
<td>C++ User’s Guide</td>
<td>Provides information on command-line options and how to use the compiler.</td>
</tr>
<tr>
<td></td>
<td>Sun WorkShop Memory Monitor User’s Manual</td>
<td>Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see /opt/SUNWspro/docs/index.html) and not at the docs.sun.com Web site.</td>
</tr>
<tr>
<td>Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95</td>
<td>Fortran Library Reference</td>
<td>Provides details about the library routines supplied with the Fortran compiler.</td>
</tr>
<tr>
<td>Document Collection</td>
<td>Document Title</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Fortran Programming Guide</strong></td>
<td>Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.</td>
<td></td>
</tr>
<tr>
<td><strong>Fortran User's Guide</strong></td>
<td>Provides information on command-line options and how to use the compilers.</td>
<td></td>
</tr>
<tr>
<td><strong>FORTTRAN 77 Language Reference</strong></td>
<td>Provides a complete language reference.</td>
<td></td>
</tr>
<tr>
<td><strong>Interval Arithmetic Programming Reference</strong></td>
<td>Describes the intrinsic INTERVAL data type supported by the Fortran 95 compiler.</td>
<td></td>
</tr>
<tr>
<td><strong>Forte™ TeamWare 6 / Sun WorkShop TeamWare 6 User's Guide</strong></td>
<td>Describes how to use the Sun WorkShop TeamWare code management tools.</td>
<td></td>
</tr>
<tr>
<td><strong>Forte™ / Sun Performance Library 6</strong></td>
<td><strong>Sun Performance Library Reference</strong></td>
<td>Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.</td>
</tr>
<tr>
<td><strong>Forte™ / Sun Performance Library User's Guide</strong></td>
<td>Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.</td>
<td></td>
</tr>
<tr>
<td><strong>Numerical Computation Guide</strong></td>
<td><strong>Numerical Computation Guide</strong></td>
<td>Describes issues regarding the numerical accuracy of floating-point computations.</td>
</tr>
<tr>
<td><strong>Standard Library 2</strong></td>
<td><strong>Standard C++ Class Library Reference</strong></td>
<td>Provides details on the Standard C++ Library.</td>
</tr>
<tr>
<td><strong>Standard C++ Library User's Guide</strong></td>
<td>Describes how to use the Standard C++ Library.</td>
<td></td>
</tr>
<tr>
<td><strong>Tools.h++ 7</strong></td>
<td><strong>Tools.h++ Class Library Reference</strong></td>
<td>Provides details on the Tools.h++ class library.</td>
</tr>
<tr>
<td><strong>Tools.h++ User's Guide</strong></td>
<td>Discusses use of the C++ classes for enhancing the efficiency of your programs.</td>
<td></td>
</tr>
</tbody>
</table>
TABLE P-4 describes related Solaris documentation available through the docs.sun.com Web site.

<table>
<thead>
<tr>
<th>Document Collection</th>
<th>Document Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris Software Developer</td>
<td><em>Linker and Libraries Guide</em></td>
<td>Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.</td>
</tr>
<tr>
<td></td>
<td><em>Programming Utilities Guide</em></td>
<td>Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.</td>
</tr>
</tbody>
</table>
Introduction

The Sun Fortran compilers, f77 and f95, described in this book (and the companion book Sun Fortran User's Guide) are available under the Solaris operating environment on the various hardware platforms that Solaris supports. The compilers themselves conform to published Fortran language standards, and provide many extended features, including multiprocessor parallelization, sophisticated optimized code compilation, and mixed C/Fortran language support.

The compilers are components of the Sun Performance WorkShop™ 6. The Fortran 90 compiler, f90, of previous releases of the Sun Performance WorkShop has been renamed f95 in Sun WorkShop 6. The f90 command is now an alias for f95 — both invoke the Sun Performance WorkShop 6 Fortran 95 compiler.

Standards Conformance

- f77 was designed to be compatible with the ANSI X3.9-1978 Fortran standard and the corresponding International Organization for Standardization (ISO) 1539-1980, as well as standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- Floating-point arithmetic for both compilers is based on IEEE standard 754-1985, and international standard IEC 60559:1989.
- In this document, "Standard" means conforming to the versions of the standards listed above. "Non-standard" or "Extension" refers to features that go beyond these versions of these standards.
The responsible standards bodies may revise these standards from time to time. The versions of the applicable standards to which these compilers conform may be revised or replaced, resulting in features in future releases of the Sun Fortran compilers that create incompatibilities with earlier releases.

---

Features of the Fortran Compilers

Sun Fortran compilers provide the following features or extensions:

- **f77**: Global program checking across routines for consistency of arguments, commons, parameters, and the like.
- **SPARC only**: Support for multiprocessor systems, including automatic and explicit loop parallelization, is integrated tightly with optimization.

**Note** – Parallelization features of the Fortran compilers require a Sun WorkShop HPC license.

- **f77**: Many VAX/VMS Fortran 5.0 extensions, including:
  - NAMELIST
  - DO WHILE
  - Structures, records, unions, maps
  - Variable format expressions
  - Recursion
  - Pointers
  - Double-precision complex
  - SPARC: Quadruple-precision real
  - SPARC: Quadruple-precision complex

- Cray-style parallelization directives, including TASK COMMON, with extensions for **f95**.
- OpenMP parallelization directives accepted by **f95**.
- Global, peephole, and potential parallelization optimizations produce high performance applications. Benchmarks show that optimized applications can run significantly faster when compared to unoptimized code.
- Common calling conventions on Solaris systems permit routines written in C or C++ to be combined with Fortran programs.
- Support for 64-bit Solaris environments on UltraSPARC platforms.
- Call-by-value, %VAL, implemented in both **f77** and **f95**.
- Interoperability between Fortran 77 and Fortran 95 programs and object binaries.
- Interval Arithmetic expressions in **f95**.
Other Fortran Utilities

The following utilities provide assistance in the development of software programs in Fortran:

- **Sun WorkShop Performance Analyzer** — In depth performance analysis tool for single threaded and multi-threaded applications. See analyzer(1).
- **asa** — This Solaris utility is a Fortran output filter for printing files that have Fortran carriage-control characters in column one. Use asa to transform files formatted with Fortran carriage-control conventions into files formatted according to UNIX line-printer conventions. See asa(1).
- **fpp** — A Fortran source code preprocessor. See fpp(1).
- **fsplit** — This utility splits one Fortran file of several routines into several files, each with one routine per file. Use fsplit on FORTRAN 77 or Fortran 95 source files. See fsplit(1)

Debugging Utilities

The following debugging utilities are available:

- **error** — *(F77 only)* A utility to merge compiler error messages with the Fortran 77 source file. (This utility is included if you do a developer install, rather than an end user install of Solaris; it is also included if you install the SUNWbtool package.)
- **-Xlist** — A compiler option to check across routines for consistency of arguments, COMMON blocks, and so on.
- **Sun WorkShop** — Provides a visual debugging environment based on dbx and includes a data visualizer and performance data collector.
Sun Performance Library

The Sun Performance Library™ is a library of optimized subroutines and functions for computational linear algebra and Fourier transforms. It is based on the standard libraries LAPACK, BLAS, FFTPACK, VFFTPACK, and LINPACK.

Each subprogram in the Sun Performance Library performs the same operation and has the same interface as the standard library versions, but is generally much faster and possibly more accurate.


Interval Arithmetic

This release of the Fortran 95 compiler introduces two new compiler flags, -xia and -xinterval that enable the compiler to recognize new language extensions and generate the appropriate code to implement interval arithmetic computations.

See the Interval Arithmetic Programming Reference for details.

Man Pages

On-line manual (man) pages provide immediate documentation about a command, function, subroutine, or collection of such things.

Sun WorkShop man pages are located in install_directory/SUNWspro/man/. (In a normal install of the Sun WorkShop, install_directory is /opt.) Add this path to your MANPATH environment variable to access these Sun WorkShop man pages. (See the Fortran User’s Guide for details.)

You can display a man page by running the command:

```
demo% man topic
```
Throughout the Fortran documentation, man page references appear with the topic name and man section number: \texttt{f77(1)} is accessed with \texttt{man f77}. Other sections, denoted by \texttt{ieee\_flags(3M)} for example, are accessed using the \texttt{-s} option on the \texttt{man} command:

\begin{verbatim}
demo% man -s 3M ieee_flags
\end{verbatim}

Man pages for the Fortran library routines are found in section 3F.

The following lists man pages of interest to Fortran user:

- \texttt{f77(1)} and \texttt{f95(1)}: The Fortran compilers command-line options
- \texttt{analyzr(1)}: Sun WorkShop Performance Analyzer
- \texttt{asa(1)}: Fortran carriage-control print output post-processor
- \texttt{dbx(1)}: Command-line interactive debugger
- \texttt{fpp(1)}: Fortran source code pre-processor
- \texttt{cpp(1)}: C source code pre-processor
- \texttt{fsplit(1)}: Pre-processor splits Fortran 77 routines into single files
- \texttt{ieee\_flags(3M)}: Examine, set, or clear floating-point exception bits
- \texttt{ieee\_handler(3M)}: Handle floating-point exceptions
- \texttt{matherr(3M)}: Math library error handling routine
- \texttt{ild(1)}: Incremental link editor for object files
- \texttt{ld(1)}: Link editor for object files
READMEs

The READMEs directory contains files that describe new features, software incompatibilities, bugs, and information that was discovered after the manuals were printed. The location of this directory depends on where your software was installed. The path is: \texttt{install\_directory/SUNWspro/READMEs/}. In a normal install, \texttt{install\_directory} is \texttt{/opt}.

\begin{table}[h]
\centering
\caption{READMEs of Interest}
\begin{tabular}{|l|l|}
\hline
README File & Describes... \\
\hline
\texttt{fortran\_77} & new and changed features, known limitations, documentation errata for this release of the FORTRAN 77 compiler, \texttt{f77}. \\
\texttt{fortran\_95} & new and changed features, known limitations, documentation errata for this release of the Fortran 95 compiler, \texttt{f95}. \\
\texttt{fpp\_readme} & overview of \texttt{fpp} features and capabilities \\
\texttt{interval\_arithmetic} & overview of the interval arithmetic features in \texttt{f95} \\
\texttt{math\_libraries} & optimized and specialized math libraries available. \\
\texttt{profiling\_tools} & using the performance profiling tools, \texttt{prof}, \texttt{gprof}, and \texttt{tcov}. \\
\texttt{runtime\_libraries} & libraries and executables that can be redistributed under the terms of the End User License. \\
\texttt{64bit\_Compilers} & compiling for 64-bit Solaris operating environments. \\
\texttt{performance\_library} & overview of the Sun Performance Library \\
\hline
\end{tabular}
\end{table}

The READMEs for all compilers are easily accessed by the \texttt{-xhelp=readme} command-line option. For example, the command:

\begin{verbatim}
f95 -xhelp=readme
\end{verbatim}

will display the \texttt{fortran\_95} README file directly.
Command-Line Help

You can view very brief descriptions of the f77 and f90 command line options by invoking the compiler’s -help option as shown below:

```
% f77 -help  -or-  
f95 -help
```

Items within [ ] are optional. Items within < > are variable parameters. Bar | indicates choice of literal values. For example:
- someoption[=<yes|no>] implies -someoption is -someoption=yes

- a: Collect data for tcov basic block profiling (old format)
- arg=local: Preserve actual arguments over ENTRY statements
- autopar: Enable automatic loop parallelization (requires WorkShop license)
- Bdynamic: Allow dynamic linking
- Bstatic: Require static linking
- c: Compile only - produce .o files, suppress linking
- C: Enable runtime subscript range checking
- cg89: Generate code for generic SPARC V7 architecture
- cg92: Generate code for SPARC V8 architecture
- copyargs: Allow assignment to constant arguments
... etc.
Fortran Input/Output

This chapter discusses the input/output features provided by Sun Fortran compilers.

Accessing Files From Within Fortran Programs

Data is transferred between the program and devices or files through a Fortran logical unit. Logical units are identified in an I/O statement by a logical unit number, a nonnegative integer from 0 to the maximum 4-byte integer value (2,147,483,647).

The character * can appear as a logical unit identifier. The asterisk stands for standard input file when it appears in a READ statement; it stands for standard output file when it appears in a WRITE or PRINT statement.

A Fortran logical unit can be associated with a specific, named file through the OPEN statement. Also, certain “preconnected” units are automatically associated with specific files at the start of program execution.

Accessing Named Files

The OPEN statement’s FILE= specifier establishes the association of a logical unit to a named, physical file at runtime. This file can be pre-existing or created by the program. See the Sun FORTRAN 77 Language Reference Manual for a full discussion of the OPEN statement.
The FILE= specifier on an OPEN statement may specify a simple file name (FILE='myfile.out') or a file name preceded by an absolute or relative directory path (FILE='../Amber/Qproj/myfile.out'). Also, the specifier may be a character constant, variable, or character expression.

Library routines can be used to bring command-line arguments and environment variables into the program as character variables for use as file names in OPEN statements. (See man page entries for getarg(3F) and getenv(3F) for details; these and other useful library routines are also described in the Fortran Library Reference).

The following example (GetFilNam.f) shows one way to construct an absolute path file name from a typed-in name. The program uses the library routines GETENV, LNBLNK, and GETCWD to return the value of the $HOME environment variable, find the last non-blank in the string, and determine the current working directory:

```fortran
CHARACTER F*128, FN*128, FULLNAME*128
PRINT*, 'ENTER FILE NAME:'
READ *, F
FN = FULLNAME( F )
PRINT *, 'PATH IS: ',FN
END

CHARACTER*128 FUNCTION FULLNAME( NAME )
CHARACTER NAME*(*) , PREFIX128
C          This assumes C shell.
C          Leave absolute path names unchanged.
C          If name starts with '~/', replace tilde with home
C          directory; otherwise prefix relative path name with
C          path to current directory.
IF ( NAME(1:1) .EQ. '/' ) THEN
  FULLNAME = NAME
ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
  CALL GETENV( 'HOME', PREFIX )
  FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
                NAME(2:LNBLNK(NAME))
ELSE
  CALL GETCWD( PREFIX )
  FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
                '/' // NAME(:LNBLNK(NAME))
ENDIF
RETURN
END
```
Compiling and running `GetFilNam.f` results in:

```
demo% pwd
/home/users/auser/subdir
demo% f77 -silent -o getfil GetFilNam.f
demo% getfil
anyfile
/home/users/auser/subdir/anyfile
demo%
```

Opening Files Without a Name

The `OPEN` statement need not specify a name; the runtime system supplies a file name according to several conventions.

Opened as Scratch

Specifying `STATUS='SCRATCH'` in the `OPEN` statement 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 (with `f77`) `STATUS='KEEP'` is specified in the `CLOSE` statement.

Already Open

If the file has already been opened by the program, you can use a subsequent `OPEN` statement to change some of the file’s characteristics; for example, `BLANK` and `FORM`. In this case, you would specify only the file’s logical unit number and the parameters to change.

Preconnected Units

Three unit numbers are automatically associated with specific standard I/O files at the start of program execution. These preconnected units are `standard input`, `standard output`, and `standard error`:

- Standard input is logical unit 5  (also Fortran 95 unit 100)
- Standard output is logical unit 6  (also Fortran 95 unit 101)
- Standard error is logical unit 0  (also Fortran 95 unit 102)
Typically, standard input receives input from the workstation keyboard; standard output and standard error display output on the workstation screen.

In all other cases where a logical unit number but no FILE= name is specified on an OPEN statement, a file is opened with a name of the form fort.\textit{n}, where \textit{n} is the logical unit number.

**Opening Files Without an OPEN Statement**

Use of the OPEN statement is optional in those cases where default conventions can be assumed. If the first operation on a logical unit is an I/O statement other than OPEN or INQUIRE, the file fort.\textit{n} is referenced, where \textit{n} is the logical unit number (except for 0, 5, and 6, which have special meaning).

These files need not exist before program execution. If the first operation on the file is not an OPEN or INQUIRE statement, they are created.

Example: The WRITE in the following code creates the file fort.25 if it is the first input/output operation on that unit:

```fortran
IU=25
WRITE( IU, '(I4)' ) IU
END
```

The preceding program opens the file fort.25 and writes a single formatted record onto that file:

```bash
demo% f77 -silent -o testunit TestUnit.f
demo% testunit
demo% cat fort.25
```

**Passing File Names to Programs**

The file system does not have any automatic facility to associate a logical unit number in a Fortran program with a physical file.

However, there are several satisfactory ways to communicate file names to a Fortran program.
Via Runtime Arguments and \textit{GETARG}

The library routine \texttt{getarg(3F)} can be used to read the command-line arguments at runtime into a character variable. The argument is interpreted as a file name and used in the \texttt{OPEN} statement \texttt{FILE=} specifier:

```
demo% cat testarg.f
  CHARACTER outfile*40
  C Get first arg as output file name for unit 51
  CALL getarg(1,outfile)
  OPEN(51,FILE=outfile)
  WRITE(51,*),'Writing to file: ', outfile
END
```

```
demo% f77 -silent -o tstarg testarg.f
demo% tstarg AnyFileName
demo% cat AnyFileName
  Writing to file: AnyFileName
demo%
```

Via Environment Variables and \textit{GETENV}

Similarly, the library routine \texttt{getenv(3F)} can be used to read the value of any environment variable at runtime into a character variable that in turn is interpreted as a file name:

```
demo% cat testenv.f
  CHARACTER outfile*40
  C Get $OUTFILE as output file name for unit 51
  CALL getenv('OUTFILE',outfile)
  OPEN(51,FILE=outfile)
  WRITE(51,*),'Writing to file: ', outfile
END
```

```
demo% f77 -silent -o tstenv testenv.f
demo% setenv OUTFILE EnvFileName
demo% tstenv
demo% cat EnvFileName
  Writing to file: EnvFileName
demo%
```
When using getarg or getenv, care should be taken regarding leading or trailing blanks. (FORTRAN 77 programs can use the library function LNBLNK; Fortran 95 programs can use the intrinsic function TRIM.) Additional flexibility to accept relative path names can be programmed along the lines of the FULLNAME function in the example at the beginning of this chapter.

**£77: Logical Unit Preattachment Using IOINIT**

The library routine IOINIT can also be used with £77 to attach logical units to specific files at runtime. IOINIT looks in the environment for names of a user-specified form and then opens the corresponding logical unit for sequential formatted I/O. Names must be of the general form PREFIXnn, where the particular PREFIX is specified in the call to IOINIT, and nn is the logical unit to be opened. Unit numbers less than 10 must include the leading 0. See the Sun Fortran Library Reference, and the IOINIT(3F) man page. *(The IOINIT facility is not implemented for £95.)*

Example: Associate physical files test.inp and test.out in the current directory to logical units 1 and 2:

First, set the environment variables.

With ksh or sh:

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

With csh:

```
demo% setenv TST01 ini1.inp
demo% setenv TST02 ini1.out
```
Chapter 2  Fortran Input/Output

The program ini1.f reads 1 and writes 2:

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

demo%

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

```fortran
demo% cat ini1.inp
12 3.14159012 6
```

demo% f77 -silent -o tstinit ini1.f
demo% tstinit
demo% cat ini1.out
12 3.14159 6
demo%

IOINIT is adequate for most programs as written. However, it is written in Fortran specifically to serve as an example for similar user-supplied routines. Retrieve a copy from the following file, a part of the FORTRAN 77 package installation:

/opt/SUNWspro/<release>/src/ioinit.f, where <release> varies for each software release. (Contact your system administrator for details.)

**Command-Line I/O Redirection and Piping**

Another way to associate a physical file with a program’s logical unit number is by redirecting or piping the preconnected standard I/O files. Redirection or piping occurs on the runtime execution command.

In this way, a program that reads standard input (unit 5) and writes to standard output (unit 6) or standard error (unit 0) can, by redirection (using <, >, >>, >&, |, |&>, 2>, 2>&1 on the command line), read or write to any other named file.
This is shown in the following table:

<table>
<thead>
<tr>
<th>Action</th>
<th>Using C Shell</th>
<th>Using Bourne or Korn Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard input — read from mydata</td>
<td>myprog &lt; mydata</td>
<td>myprog &lt; mydata</td>
</tr>
<tr>
<td>Standard output — write (overwrite)</td>
<td>myprog &gt; myoutput</td>
<td>myprog &gt; myoutput</td>
</tr>
<tr>
<td>Standard output — write/append to myoutput</td>
<td>myprog &gt;&gt; myoutput</td>
<td>myprog &gt;&gt; myoutput</td>
</tr>
<tr>
<td>Redirect standard error to a file</td>
<td>myprog &gt; errorfile</td>
<td>myprog 2&gt; errorfile</td>
</tr>
<tr>
<td>Pipe standard output to input of another program</td>
<td>myprog1</td>
<td>myprog2</td>
</tr>
<tr>
<td>Pipe standard error and output to another program</td>
<td>myprog1</td>
<td>myprog2</td>
</tr>
</tbody>
</table>

See the csh, ksh, and sh man pages for details on redirection and piping on the command line.

**f77: VAX / VMS Logical File Names**

If you are porting from VMS FORTRAN to FORTRAN 77, the VMS-style logical file names in the INCLUDE statement are mapped to UNIX path names. The environment variable LOGICALNAMEMAPPING defines the mapping between the logical names and the UNIX path name. If the environment variable LOGICALNAMEMAPPING is set and the –vax, –xl or –xld compiler options are used, the compiler interprets VMS logical file names on the INCLUDE statement.

The compiler sets the environment variable to a string with the following syntax:

```
"iname1=path1; iname2=path2; ... "
```
Each *lname* is a logical name, and each *path* is the path name of a directory (without a trailing `/`). All blanks are ignored when parsing this string. Any trailing `/list` or `/nolist` is stripped from the file name in the INCLUDE statement. Logical names in a file name are delimited by the first colon in the VMS file name. The compiler converts file names of the form:

```
 lname : file
```

to:

```
 path / file
```

Uppercase and lowercase are significant in logical names. If a logical name is encountered on the INCLUDE statement that was not specified by LOGICALNAMEMAPPING, the file name is used unchanged.

---

**Direct I/O**

Direct or random I/O allows you to access a file directly by record number. Record numbers are assigned when a record is written. Unlike sequential I/O, direct I/O records can be read and written in any order. However, in a direct access file, all records must be the same fixed length. Direct access files are declared with the ACCESS='DIRECT' specifier on the OPEN statement for the file.

A logical record in a direct access file is a string of bytes of a length specified by the OPEN statement’s RECL= specifier. READ and WRITE statements must not specify logical records larger than the defined record size. (Record sizes are specified in bytes.) Shorter 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.

Direct access READ and WRITE statements have an extra argument, REC=*, to specify the record number to be read or written.

Example: Direct access, unformatted:

```
 OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=200, &
     FORM='UNFORMATTED', ERR=90 )
 READ( 2, REC=13, ERR=30 ) X, Y
```
This program opens a file for direct access, unformatted I/O, with a fixed record length of 200 bytes, then reads the thirteenth record into X and Y.

Example: Direct access, formatted:

```fortran
OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=200,
      FORM='FORMATTED', ERR=90 )
READ( 2, FMT='(I10,F10.3)', REC=13, ERR=30 ) X, Y
```

This program opens a file for direct access, formatted I/O, with a fixed record length of 200 bytes. It then reads the thirteenth record and converts it with the format (I10,F10.3).

For formatted files, the size of the record written is determined by the FORMAT statement. In the preceding example, the FORMAT statement defines a record of 20 characters or bytes. More than one record can be written by a single formatted write if the amount of data on the list is larger than the record size specified in the FORMAT statement. In such a case, each subsequent record is given successive record numbers.

Example: Direct access, formatted, multiple record write:

```fortran
OPEN( 21, ACCESS='DIRECT', RECL=200, FORM='FORMATTED')
WRITE(21,'(10F10.3)',REC=11) (X(J),J=1,100)
```

The write to direct access unit 21 creates 10 records of 10 elements each (since the format specifies 10 elements per record) these records are numbered 11 through 20.

## Binary I/O

Sun Workshop Fortran 95 and Fortran 77 extend the OPEN statement to allow declaration of a “binary” I/O file.

Opening a file with FORM='BINARY' has roughly the same effect as FORM='UNFORMATTED', except that no record lengths are embedded in the file. Without this data, there is no way to tell where one record begins, or ends. Thus, it is impossible to BACKSPACE a FORM='BINARY' file, because there is no way of telling where to backspace to. A READ on a 'BINARY' file will read as much data as needed to fill the variables on the input list.

- WRITE statement: Data is written to the file in binary, with as many bytes transferred as specified by the output list.
■ **READ** statement: Data is read into the variables on the input list, transferring as many bytes as required by the list. Because there are no record marks on the file, there will be no “end-of-record” error detection. The only errors detected are “end-of-file” or abnormal system errors.

■ **INQUIRE** statement: **INQUIRE** on a file opened with **FORM=“BINARY”** returns:

- **FORM=“BINARY”**
- **ACCESS=“SEQUENTIAL”**
- **DIRECT=“NO”**
- **FORMATTED=“NO”**
- **UNFORMATTED=“YES”**
- **RECL=** and **NEXTREC=** are undefined

■ **BACKSPACE** statement: Not allowed—returns an error.

■ **ENDFILE** statement: Truncates file at current position, as usual.

■ **REWIND** statement: Repositions file to beginning of data, as usual.

---

**Internal Files**

An internal file is an object of type **CHARACTER** such as a variable, substring, array, element of an array, or field of a structured record. Internal file **READ** can be from a constant character string. **I/O** on internal files simulates formatted **READ** and **WRITE** statements by transferring and converting data from one character object to another data object. No file **I/O** is performed.

When using internal files:

■ The name of the character object receiving the data appears in place of the unit number on a **WRITE** statement. On a **READ** statement, the name of the character object source appears in place of the unit number.

■ A constant, variable, or substring object constitutes a single record in the file.

■ With an array object, each array element corresponds to a record.

■ **£77**: £77 extends direct **I/O** to internal files. (The ANSI standard includes only sequential formatted **I/O** on internal files.) This is similar to direct **I/O** on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings.

■ Each sequential **READ** or **WRITE** statement starts at the beginning of an internal file.
Example: Sequential formatted read from an internal file (one record only):

```fortran
demo% cat intern1.f
CHARACTER X*80
READ( *, '(A)' ) X
READ( X, '(I3,I4)' ) N1, N2 ! This codeline reads the internal file X
WRITE( *, '*' ) N1, N2
END
demo% f77 -silent -o tstintern intern1.f
demo% tstintern
12 99
12 99
demo% 
```

Example: Sequential formatted read from an internal file (three records):

```fortran
demo% cat intern2.f
CHARACTER LINE(4)*16 ! This is our “internal file”
*
12341234
DATA LINE(1) / ' 81  81 ' /
DATA LINE(2) / ' 82  82 ' /
DATA LINE(3) / ' 83  83 ' /
DATA LINE(4) / ' 84  84 ' /
READ( LINE,'(2I4)' ) I,J,K,L,M,N
PRINT *, I, J, K, L, M, N
END
demo% f77 -silent intern2.f
demo% a.out
81  81  82  82  83  83
```
Example: Direct access read from an internal file (one record) (*f77 only):

```fortran
CHARACTER LINE(4)*16  ! This is our “internal file”
   12341234
 DATA  LINE(1) / ' 81  81 ' /
 DATA  LINE(2) / ' 82  82 ' /
 DATA  LINE(3) / ' 83  83 ' /
 DATA  LINE(4) / ' 84  84 ' /
READ ( LINE, FMT=20, REC=3 ) M, N
20       FORMAT( I4, I4 )
 PRINT *, M, N
END
```

**f77: Tape I/O**

Most typical Fortran I/O is done to disk files. However, by associating a logical unit number to a physically mounted tape drive via the OPEN statement, it is possible to do I/O directly to tape.

It could be more efficient to use the `TOPEN()` routines rather than Fortran I/O statements to do I/O on magnetic tape.

**Using TOPEN Routines**

With the nonstandard tape I/O package (see `topen(3F)`) you can transfer blocks between the tape drive and buffers declared as Fortran character variables. You can then use internal I/O to fill and empty these buffers. This facility does not integrate with the rest of Fortran I/O and even has its own set of tape logical units. Refer to the man pages for complete information.
Fortran Formatted I/O for Tape

The Fortran I/O statements provide facilities for transparent access to formatted, sequential files on magnetic tape. There is no limit on formatted record size, and records may span tape blocks.

Fortran Unformatted I/O for Tape

Using the Fortran I/O statements to connect a magnetic tape for unformatted access is less satisfactory. The implementation of unformatted records implies that the size of a record (plus eight characters of overhead) cannot be bigger than the buffer size.

As long as this restriction is complied with, the I/O system does not write records that span physical tape blocks, writing short blocks when necessary. This representation of unformatted records is preserved (even though it is inappropriate for tapes) so that files can be freely copied between disk and tapes.

Since the block-spanning restriction does not apply to tape reads, files can be copied from tape to disk without any special considerations.

Tape File Representation

A Fortran data file is represented on tape by a sequence of data records followed by an endfile record. The data is grouped into blocks, with maximum block size determined when the file is opened. The records are represented in the same way as records in disk files: formatted records are followed by newlines; unformatted records are preceded and followed by character counts. In general, there is no relation between Fortran records and tape blocks; that is, records can span blocks, which can contain parts of several records.

The only exception is that Fortran does not write an unformatted record that spans blocks; thus, the size of the largest unformatted record is eight characters less than the block size.

The dd Conversion Utility

An end-of-file record in Fortran maps directly into a tape mark. In this respect, Fortran files are the same as tape system files. But since the representation of Fortran files on tape is the same as that used in the rest of UNIX, naive Fortran programs cannot read 80-column card images on tape. If you have an existing Fortran program and an existing data tape to read with it, translate the tape using the dd(1) utility, which adds newlines and strips trailing blanks.
Example: Convert a tape on mt0 and pipe that to the executable ftnprg:

```
demo% dd if=/dev/rmt0 ibs=20b cbs=80 conv=unblock | ftnprg
```

The `getc` Library Routine

As an alternative to `dd`, you can call the `getc(3F)` library routine to read characters from the tape. You can then combine the characters into a character variable and use internal I/O to transfer formatted data. See also `OPEN(3F)`.

End-of-File

The end-of-file condition is reached when an end-of-file record is encountered during execution of a READ statement. The standard states that the file is positioned after the end-of-file record. In real life, this means that the tape read head is poised at the beginning of the next file on the tape. Although it seems as if you could read the next file on the tape, this is not strictly true, and is not covered by the ANSI FORTRAN 77 Language Standard.

The standard also says that a `BACKSPACE` or `REWIND` statement can be used to reposition the file. Consequently, after reaching end-of-file, you can backspace over the end-of-file record and further manipulate the file—for example, writing more records at the end, rewinding the file, and rereading or rewriting it.

Multifile Tapes

The name used to open the tape file determines certain characteristics of the connection, such as the recording density and whether the tape is automatically rewound when opened and closed.

To access a file on a tape with multiple files, first use the `mt(1)` utility to position the tape to the needed file. Then open the file as a no-rewind magnetic tape such as `/dev/nrmt0`. Referencing the tape with this name prevents it from being repositioned when it is closed. By reading the file until end-of-file and then reopening it, a program can access the next file on the tape. Any program subsequently referencing the same tape can access it where it was last left, preferably at the beginning of a file, or past the end-of-file record.

However, if your program terminates prematurely, it may leave the tape positioned anywhere. Use the SunOS™ operating system command `mt(1)` to reposition the tape appropriately.
Fortran 95 I/O Considerations

Sun WorkShop 6 Fortran 95 and Fortran 77 are I/O compatible. Executables containing intermixed $f77$ and $f95$ compilations can do I/O to the same unit from both the $f77$ and $f95$ parts of the program.

However, Fortran 95 provides some additional features:

- **ADVANCE='NO'** enables nonadvancing I/O, as in:

  ```fortran
  write(*,'(a)',ADVANCE='NO')  'Enter size= '
  read(*,*)  n
  ```

- **NAMELIST** input features:
  - $f95$ allows the group name to be preceded by $\&$ or $\&$ on input. The Fortran 95 standard accepts only $\&$ and this is what a NAMELIST write outputs.
  - $f95$ accepts $\&$ as the symbol terminating an input group unless the last data item in the group is CHARACTER, in which case the $\&$ is treated as input data.
  - $f95$ allows NAMELIST input to start in the first column of a record.
  - **ENCODE** and **DECODE** are recognized and implemented by $f95$ just as they are by $f77$.```
Program Development

This chapter briefly introduces two powerful program development tools, *make* and SCCS, that can be used very successfully with Fortran programs.

A number of good, commercially published books on using *make* and SCCS are currently available, including *Managing Projects with make*, by Andrew Oram and Steve Talbott, and *Applying RCS and SCCS*, by Don Bolinger and Tan Bronson. Both are from O'Reilly & Associates.

Facilitating Program Builds With the make Utility

The *make* utility applies intelligence to the task of program compilation and linking. Typically, a large application consists of a set of source files and INCLUDE files, requiring linking with a number of libraries. Modifying any one or more of the source files requires recompilation of that part of the program and relinking. You can automate this process by specifying the interdependencies between files that make up the application along with the commands needed to recompile and relink each piece. With these specifications in a file of directives, *make* ensures that only the files that need recompiling are recompiled and that relinking uses the options and libraries you need to build the executable. The following discussion provides a simple example of how to use *make*. For a summary, see *make*(1).

The Makefile

A file called *makefile* tells *make* in a structured manner which source and object files depend on other files. It also defines the commands required to compile and link the files.
For example, suppose you have a program of four source files and the makefile:

```
demo% ls
makefile
commonblock
computepts.f
pattern.f
startupcore.f
```

Assume both `pattern.f` and `computepts.f` have an INCLUDE of `commonblock`, and you wish to compile each `.f` file and link the three relocatable files, along with a series of libraries, into a program called `pattern`.

The makefile looks like this:

```
demo% cat makefile
pattern: pattern.o computepts.o startupcore.o
    f77 pattern.o computepts.o startupcore.o -lcore77 -lcore -lsunwindow -lpixrect -o pattern
pattern.o: pattern.f commonblock
    f77 -c -u pattern.f
computepts.o: computepts.f commonblock
    f77 -c -u computepts.f
startupcore.o: startupcore.f
    f77 -c -u startupcore.f
demo%
```

The first line of `makefile` indicates that making `pattern` depends on `pattern.o`, `computepts.o`, and `startupcore.o`. The next line and its continuations give the command for making `pattern` from the relocatable `.o` files and libraries.

Each entry in `makefile` is a rule expressing a target object’s dependencies and the commands needed to make that object. The structure of a rule is:

```
target: dependencies-list
TAB build-commands
```

- **Dependencies.** Each entry starts with a line that names the target file, followed by all the files the target depends on.
- **Commands.** Each entry has one or more subsequent lines that specify the Bourne shell commands that will build the target file for this entry. Each of these command lines must be indented by a tab character.
make Command

The make command can be invoked with no arguments, simply:

```
demo$ make
```

The make utility looks for a file named makefile or Makefile in the current directory and takes its instructions from that file.

The make utility:
- Reads makefile to determine all the target files it must process, the files they depend on, and the commands needed to build them.
- Finds the date and time each file was last changed.
- Rebuilds any target file that is older than any of the files it depends on, using the commands from makefile for that target.

Macros

The make utility’s macro facility allows simple, parameterless string substitutions. For example, the list of relocatable files that make up the target program pattern can be expressed as a single macro string, making it easier to change.

A macro string definition has the form:

```
NAME = string
```

Use of a macro string is indicated by:

```
$(NAME)
```

which is replaced by make with the actual value of the macro string.

This example adds a macro definition naming all the object files to the beginning of makefile:

```
OBJ = pattern.o computepts.o startupcore.o
```
Now the macro can be used in both the list of dependencies as well as on the `f77` link command for target `pattern` in `makefile`:

```plaintext
pattern: $(OBJ)
f77 $(OBJ) -lcore77 -lcore -lsunwindow 
   -lpixrect -o pattern
```

For macro strings with single-letter names, the parentheses may be omitted.

**Overriding of Macro Values**

The initial values of `make` macros can be overridden with command-line options to `make`. For example:

```plaintext
FFLAGS=-u
OBJ = pattern.o computepts.o startupcore.o
pattern: $(OBJ)
f77 $(OBJ) -lcore77 -lcore -lsunwindow 
   -lpixrect -o pattern
pattern.o: pattern.f commonblock
   f77 $(FFLAGS) -c pattern.f
computepts.o: computepts.f
   f77 $(FFLAGS) -c computepts.f
```

Now a simple `make` command without arguments uses the value of `FFLAGS` set above. However, this can be overridden from the command line:

```plaintext
demo% make "FFLAGS=-u -O"
```

Here, the definition of the `FFLAGS` macro on the `make` command line overrides the `makefile` initialization, and both the `-O` flag and the `-u` flag are passed to `f77`. Note that "`FFLAGS=-" can also be used on the command to reset the macro to a null string so that it has no effect.

**Suffix Rules in `make`**

To make writing a `makefile` easier, `make` will use its own default rules depending on the suffix of a target file. Recognizing the `.f` suffix, `make` uses the `f77` compiler, passing as arguments any flags specified by the `FFLAGS` macro, the `-c` flag, and the name of the source file to be compiled.
The example below demonstrates this rule twice:

```plaintext
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
    f77 $(OBJ) -lcore77 -lcore -lsunwindow 
    -lpixrect -o pattern
pattern.o: pattern.f commonblock
    f77 $(FFLAGS) -c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

make uses default rules to compile computepts.f and startupcore.f.

Similarly, suffix rules for .f90 files will also invoke the f95 compiler. However, there are no suffix rules currently defined for .f95 Fortran 95 source files or .mod Fortran 95 module files.

---

**Version Tracking and Control With SCCS**

SCCS stands for *Source Code Control System*. SCCS provides a way to:

- Keep track of the evolution of a source file—its change history
- Prevent a source file from being simultaneously changed by other developers
- 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
- Checking in a file

This section shows you how to use SCCS to perform these tasks, using the previous program as an example. Only basic SCCS is described and only three SCCS commands are introduced: create, edit, and delget.

---

**Controlling Files With SCCS**

Putting files under SCCS control involves:

- Making the SCCS directory
- Inserting SCCS ID keywords into the files (this is optional)
- 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. Use this command:

```
demo% mkdir SCCS
```

SCCS must be in uppercase.

Inserting SCCS ID Keywords

Some developers put one or more SCCS ID keywords into each file, but that is optional. These keywords are later identified with a version number each time the files are checked in with an SCCS get or delget command. There are three likely places to put these strings:

- Comment lines
- Parameter statements
- Initialized data

The advantage of using keywords is that the version information appears in the source listing and compiled object program. If preceded by the string @(#), the keywords in the object file can be printed using the what command.

Included header files that contain 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. In some files, like ASCII data files or makefiles, the SCCS information will appear in comments.

SCCS keywords appear in the form %keyword% and are expanded into their values by the SCCS get command. The most commonly used keywords are:

- %Z% expands to the identifier string @(#) recognized by the what command.
- %M% expands to the name of the source file.
- %I% expands to the version number of this SCCS maintained file.
- %E% expands to the current date.

For example, you could identify the makefile with a make comment containing these keywords:

```
# %Z%M% %I% %E%
```
The source files, `startupcore.f`, `computepts.f`, and `pattern.f`, can be identified by initialized data of the form:

```
CHARACTER*50 SCCSID
DATA SCCSID/
  "%Z%M% %I% %E%n"
/
```

When this file is processed by SCCS, compiled, and the object file processed by the SCCS what command, the following is displayed:

```
demo% f77 -c pattern.f
...
demo% what pattern
pattern:
  pattern.f 1.2 96/06/10
```

You can also create a PARAMETER named CTIME that is automatically updated whenever the file is accessed with `get`:

```
CHARACTER*(*) CTIME
PARAMETER ( CTIME="%E%")
```

INCLUDE files can be annotated with a Fortran comment containing the SCCS stamp:

```
C  %Z%M% %I% %E%
```

**Note** – Use of single letter derived type component names in Fortran 95 source code files can conflict with SCCS keyword recognition. For example, the Fortran 95 structure component reference `X%Y%Z` when passed through SCCS will become `XZ` after an SCCS `get`. Care should be taken not to define structure components with single letters when using SCCS on Fortran 95 programs. For example, had the structure reference in the Fortran 95 program been to `X%YY%Z`, the `YY%` would not have been interpreted by SCCS as a keyword reference. Alternatively, the SCCS `get -k` option will retrieve the file without expanding SCCS keyword IDs.
Creating SCCS Files

Now you can put these files under control of SCCS with the SCCS `create` command:

```bash
demo% sccs create makefile commonblock startupcore.f \
       computepts.f pattern.f
demo%
```

Checking Files Out and In

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 have finished editing.

Check out a file with the `sccs edit` command. For example:

```bash
demo% sccs edit computepts.f
```

SCCS then makes a writable copy of `computepts.f` in the current directory, and records your login name. Other users cannot check the file out while you have it checked out, but they can find out who has checked it out.

When you have completed your editing, check in the modified file with the `sccs delget` command. For example:

```bash
demo% sccs delget computepts.f
```

This command causes the SCCS system to:

- Make sure that you are the user who checked out the file by comparing login names
- Prompt for a comment from you on the changes
- Make a record of what was changed in this editing session
- Delete the writable copy of `computepts.f` from the current directory
- Replace it by a read-only copy with the SCCS keywords expanded

The `sccs delget` command is a composite of two simpler SCCS commands, `delta` and `get`. The `delta` command performs the first three tasks in the list above; the `get` command performs the last two tasks.
Libraries

This chapter describes how to use and create libraries of subprograms. Both static and dynamic libraries are discussed.

Understanding Libraries

A software library is usually a set of subprograms that have been previously compiled and organized into a single binary library file. Each member of the set is called a library element or module. The linker searches the library files, loading object modules referenced by the user program while building the executable binary program. See `ld(1)` and the Solaris Linker and Libraries Guide for details.

There are two basic kinds of software libraries:

- **Static library.** A library in which modules are bound into the executable file before execution. Static libraries are commonly named `libname.a`. The `.a` suffix refers to archive.

- **Dynamic library.** A library in which modules can be bound into the executable program at runtime. Dynamic libraries are commonly named `libname.so`. The `.so` suffix refers to shared object.

Typical system libraries that have both static (.a) and dynamic (.so) versions are:

- **Fortran 77 libraries:** `libF77`, `libM77`
- **Fortran95 libraries:** `libfsu`, `libfui`, `libfai`, `libfai2`, `libfsunai`, `libfprodai`, `libfminlai`, `libfmaxlai`, `libminvai`, `libmaxvai`, `libf77compat`
- **VMS Fortran libraries:** `libv77`
- **C libraries:** `libc`

There are two advantages to the use of libraries:

- There is no need to have source code for the library routines that a program calls.
- Only the needed modules are loaded.
Library files provide an easy way for programs to share commonly used subroutines. You need only name the library when linking the program, and those library modules that resolve references in the program are linked and merged into the executable file.

### Specifying Linker Debugging Options

Summary information about library usage and loading can be obtained by passing additional options to the linker through the `LD_OPTIONS` environment variable. The compiler calls the linker with these options (and others it requires) when generating object binary files.

Using the compiler to call the linker is always recommended over calling the linker directly because many compiler options require specific linker options or library references, and linking without these could produce unpredictable results.

Example: Using `LD_OPTIONS` to create a load map:

```
demo% setenv LD_OPTIONS '–m –Dfiles'
demo% f77 –o myprog myprog.f
```

Some linker options do have compiler command-line equivalents that can appear directly on the `f77` or `f95` command. These include `–Bx`, `–dx`, `–G`, `–hname`, `–Rpath`, and `–ztext`. See the `f77(1)` and `f95(1)` man pages or the *Fortran User’s Guide* for details.

More detailed examples and explanations of linker options and environment variables can be found in the Solaris *Linker and Libraries Guide*.

### Generating a Load Map

The linker `–m` option generates a load map that displays library linking information. The routines linked during the building of the executable binary program are listed together with the libraries that they come from.
Example: Using \texttt{\textasciitilde m} to generate a load map:

```bash
demo% setenv LD_OPTIONS \textquoteleft\textasciitilde m\textquoteright
demo% f77 any.f
any.f:
  MAIN:

    LINK EDITOR MEMORY MAP

<table>
<thead>
<tr>
<th>output</th>
<th>input</th>
<th>virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>section</td>
<td>section</td>
<td>address</td>
</tr>
<tr>
<td>.interp</td>
<td>100d4</td>
<td>11</td>
</tr>
<tr>
<td>.interp 100d4</td>
<td>11 (null)</td>
<td></td>
</tr>
<tr>
<td>.hash</td>
<td>100e8</td>
<td>2e8</td>
</tr>
<tr>
<td>.hash 100e8</td>
<td>2e8 (null)</td>
<td></td>
</tr>
<tr>
<td>.dynsym</td>
<td>103d0</td>
<td>650</td>
</tr>
<tr>
<td>.dynsym 103d0</td>
<td>650 (null)</td>
<td></td>
</tr>
<tr>
<td>.dynstr</td>
<td>10a20</td>
<td>366</td>
</tr>
<tr>
<td>.dynstr 10a20</td>
<td>366 (null)</td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td>10c90</td>
<td>1e70</td>
</tr>
<tr>
<td>.text 10c90</td>
<td>00 /opt/SUNWspro/lib/crti.o</td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td>10c90</td>
<td>f4 /opt/SUNWspro/lib/crt1.o</td>
</tr>
<tr>
<td>.text</td>
<td>10d84</td>
<td>00 /opt/SUNWspro/lib/values-xi.o</td>
</tr>
<tr>
<td>.text</td>
<td>10d88</td>
<td>d20 sparse.o</td>
</tr>
</tbody>
</table>

...```

Listing Other Information

Additional linker debugging features are available through the linker’s \texttt{\textasciitilde Dkeyword} option. A complete list can be displayed using \texttt{\textasciitilde Dhelp}.

Example: List linker debugging aid options using the \texttt{\textasciitilde Dhelp} option:

```bash
demo% ld \textasciitilde Dhelp
... debug: args display input argument processing
default: bindings display symbol binding;
default: detail provide more information
default: entry display entrance criteria descriptors
... demo%
```

For example, the \texttt{\textasciitilde Dfiles} linker option lists all the files and libraries referenced during the link process:
Consistent Compiling and Linking

Ensuring a consistent choice of compiling and linking options is critical whenever compilation and linking are done in separate steps. Compiling any part of a program with any of the following options requires linking with the same options:

- \( -a \), \( -autopar \), \( -Bx \), \( -fast \), \( -G \), \( -Lpath \), \( -lname \), \( -mt \), \( -xmemalign \), \( -nolib \), \( -norunpath \), \( -p \), \( -pg \), \( -xlibmopt \), \( -xlic_lib=name \), \( -xprofile=p \)

Example: Compiling \( sbr.f \) with \( -a \) and \( smain.f \) without it, then linking in separate steps (\( -a \) invokes \( tcov \) old style profiling):

```
    demo% setenv LD_OPTIONS '~-Dfiles'
    demo% f77 direct.f
    direct.f:
    MAIN direct:
    debug: file=/opt/SUNWspro/lib/crti.o [ ET_REL ]
    debug: file=/opt/SUNWspro/lib/crtl.o [ ET_REL ]
    debug: file=/opt/SUNWspro/lib/values-xi.o [ ET_REL ]
    debug: file=direct.o [ ET_REL ]
    debug: file=/opt/SUNWspro/lib/libM77.a [ archive ]
    debug: file=/opt/SUNWspro/lib/libF77.so [ ET_DYN ]
    debug: file=/opt/SUNWspro/lib/libsunmath.a [ archive ]
    ...
```

```
    demo% f77 -c -a sbr.f
    demo% f77 -c smain.f
    demo% f77 -a sbr.o smain.o [link step: pass -a to the linker]
```

Also, a number of options require that \( all \) source files be compiled with that option. These include:

- \( -aligncommon \), \( -autopar \), \( -dx \), \( -dalign \), \( -dbl \), \( -explicitpar \), \( -f \), \( -misalign \), \( -native \), \( -parallel \), \( -pentium \), \( -xarch=a \), \( -xcache=c \), \( -xchip=c \), \( -xF \), \( -xtarget=f \), \( -ztext \)

See the \( f77(1) \) and \( f95(1) \) man pages and the \( Fortran User's Guide \) for details on all compiler options.
Setting Library Search Paths and Order

The linker searches for libraries at several locations and in a certain prescribed order. Some of these locations are standard paths, while others depend on the compiler options `-R path`, `-l library`, and `-L dir` and the environment variable `LD_LIBRARY_PATH`.

Search Order for Standard Library Paths

The standard library search paths used by the linker are determined by the installation path, and they differ for static and dynamic loading. `<install-point>` is the path to where the Fortran compilers have been installed. In a standard install of the software this is `/opt`.

Static Linking

While building the executable file, the static linker searches for any libraries in the following paths (among others), in the specified order:

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;install-point&gt;/SUNWspro/lib</code></td>
<td>Sun shared libraries</td>
</tr>
<tr>
<td><code>/usr/ccs/lib/</code></td>
<td>Standard location for SVr4 software</td>
</tr>
<tr>
<td><code>/usr/lib</code></td>
<td>Standard location for UNIX software</td>
</tr>
</tbody>
</table>

These are the default paths used by the linker.

Dynamic Linking

The dynamic linker searches for shared libraries at runtime, in the specified order:

- Paths specified by user with `-R path`
- `<install-point>/SUNWspro/lib/`
- `/usr/lib` standard UNIX default

The search paths are built into the executable.
LD_LIBRARY_PATH Environment Variable

Use the LD_LIBRARY_PATH environment variable to specify directory paths that the linker should search for libraries specified with the -l library option.

Multiple directories can be specified, separated by a colon. Typically, the LD_LIBRARY_PATH variable contains two lists of colon-separated directories separated by a semicolon:

    dirlist1;dirlist2

The directories in dirlist1 are searched first, followed by any explicit -Ldir directories specified on the command line, followed by dirlist2 and the standard directories.

That is, if the compiler is called with any number of occurrences of -L, as in:

    f77 ... -Lpath1 ... -Lpathn ... 

then the search order is:

    dirlist1 path1 ... pathn dirlist2 standard_paths

When the LD_LIBRARY_PATH variable contains only one colon-separated list of directories, it is interpreted as dirlist2.

In the Solaris operating environment, a similar environment variable, LD_LIBRARY_PATH_64 can be used to override LD_LIBRARY_PATH when searching for 64-bit dependencies. See the Solaris Linker and Libraries Guide and the ld(1) man page for details.

- On a 32-bit SPARC processor, LD_LIBRARY_PATH_64 is ignored.
- If only LD_LIBRARY_PATH is defined, it is used for both 32-bit and 64-bit linking.
- If both LD_LIBRARY_PATH and LD_LIBRARY_PATH_64 are defined, 32-bit linking will be done using LD_LIBRARY_PATH, and 64-bit linking with LD_LIBRARY_PATH_64.

Note – Use of the LD_LIBRARY_PATH environment variable with production software is strongly discouraged. Although useful as a temporary mechanism for influencing the runtime linker’s search path, any dynamic executable that can reference this environment variable will have its search paths altered. You might see unexpected results or a degradation in performance.
Library Search Path and Order—Static Linking

Use the `-llibrary` compiler option to name additional libraries for the linker to search when resolving external references. For example, the option `-lmylib` adds the library `libmylib.so` or `libmylib.a` to the search list.

The linker looks in the standard directory paths to find the additional `libmylib` library. The `-L` option (and the `LD_LIBRARY_PATH` environment variable) creates a list of paths that tell the linker where to look for libraries outside the standard paths.

Were `libmylib.a` in directory `/home/proj/libs`, then the option `-L/home/proj/libs` would tell the linker where to look when building the executable:

```
demo% f77 -o pgram part1.o part2.o -L/home/proj/libs -lmylib
```

Command-Line Order for `-llibrary` Options

For any particular unresolved reference, libraries are searched only once, and only for symbols that are undefined at that point in the search. If you list more than one library on the command line, then the libraries are searched in the order in which they are found on the command line. Place `-llibrary` options as follows:

- Place the `-llibrary` option after any `.f`, `.for`, `.F`, `.f95`, or `.o` files.
- If you call functions in `libx`, and they reference functions in `liby`, then place `-lx` before `-ly`.

Command-Line Order for `-Ldir` Options

The `-Ldir` option adds the `dir` directory path to the library search list. The linker searches for libraries first in any directories specified by the `-L` options and then in the standard directories. This option is useful only if it is placed preceding the `-llibrary` options to which it applies.

Library Search Path and Order—Dynamic Linking

With dynamic libraries, changing the library search path and order of loading differs from the static case. Actual linking takes place at runtime rather than build time.
Specifying Dynamic Libraries at Build Time

When building the executable file, the linker records the paths to shared libraries in the executable itself. These search paths can be specified using the -R path option. This is in contrast to the -L dir option which indicates at buildtime where to find the library specified by a -l library option, but does not record this path into the binary executable.

The directory paths that were built in when the executable was created can be viewed using the dump command.

Example: List the directory paths built into a.out:

```bash
demo% f77 program.f -R/home/proj/libs -L/home/proj/libs -lmylib
demo% dump -Lv a.out | grep RPATH
[5]      RPATH    /home/proj/libs:/opt/SUNWsp0/lib
```

Specifying Dynamic Libraries at Runtime

At runtime, the linker determines where to find the dynamic libraries that an executable needs from:

- The value of LD_LIBRARY_PATH at runtime
- The paths that had been specified by -R at the time the executable file was built

As noted earlier, use of LD_LIBRARY_PATH can have unexpected side-effects and is not recommended.

Fixing Errors During Dynamic Linking

When the dynamic linker cannot locate a needed library, it issues this error message:

```
ld.so: prog: fatal: libmylib.so: can’t open file:
```

The message indicates that the libraries are not where they are supposed to be. Perhaps you specified paths to shared libraries when the executable was built, but the libraries have subsequently been moved. For example, you might have built a.out with your own dynamic libraries in /my/libs/, and then later moved the libraries to another directory.
Use `ldd` to determine where the executable expects to find the libraries:

```
$ ldd a.out
  libsolib.so => /export/home/proj/libsolib.so
  libF77.so.4 => /opt/SUNWspro/lib/libF77.so.4
  libc.so.1 => /usr/lib/libc.so.1
  libdl.so.1 => /usr/lib/libdl.so.1
```

If possible, move or copy the libraries into the proper directory or make a soft link to the directory (using `ln -s`) in the directory that the linker is searching. Or, it could be that `LD_LIBRARY_PATH` is not set correctly. Check that `LD_LIBRARY_PATH` includes the path to the needed libraries at runtime.

---

### Creating Static Libraries

Static library files are built from precompiled object files (.o files) using the `ar(1)` utility.

The linker extracts from the library any elements whose entry points are referenced within the program it is linking, such as a subprogram, entry name, or `COMMON` block initialized in a `BLOCKDATA` subprogram. These extracted elements (routines) are bound permanently into the `a.out` executable file generated by the linker.

---

### Tradeoffs for Static Libraries

There are three main issues to keep in mind regarding static, as compared to dynamic, libraries and linking:

- **Static libraries are more self-contained but less adaptable.**

  If you bind an `a.out` executable file *statically*, the library routines it needs become part of the executable binary. However, if it becomes necessary to update a static library routine bound into the `a.out` executable, the entire `a.out` file must be relinked and regenerated to take advantage of the updated library. With *dynamic* libraries, the library is not part of the `a.out` file and linking is done at runtime. To take advantage of an updated dynamic library, all that is required is that the new library be installed on the system.

- **The “elements” in a static library are individual compilation units, .o files.**
Since a single compilation unit (a source file) can contain more than one subprogram, these routines when compiled together become a single module in the static library. This means that all the routines in the compilation unit are loaded together into the a.out executable, even though only one of those subprograms was actually called. This situation can be improved by optimizing the way library routines are distributed into compilable source files. (Still, only those library modules actually referenced by the program are loaded into the executable.)

Order matters when linking static libraries.

The linker processes its input files in the order in which they appear on the command line—left to right. When the linker decides whether or not to load an element from a library, its decision is determined by the library elements that it has already processed. This order is not only dependent on the order of the elements as they appear in the library file but also on the order in which the libraries are specified on the compile command line.

Example: If the Fortran program is in two files, main.f and crunch.f, and only the latter accesses a library, it is an error to reference that library before crunch.f or crunch.o:

incorrect commands:
```
demo% f77 main.f -lmylibrary crunch.f -o myprog
```
```
demo% f77 main.f crunch.f -lmylibrary -o myprog
```
(correct)

Creation of a Simple Static Library

Suppose that you can distribute all the routines in a program over a group of source files and that these files are wholly contained in the subdirectory test_lib/.

Suppose further that the files are organized in such a way that they each contain a single principal subprogram that would be called by the user program, along with any “helper” routines that the subprogram might call but that are called from no other routine in the library. Also, any helper routines called from more than one library routine are gathered together into a single source file. This gives a reasonably well-organized set of source and object files.
Assume that the name of each source file is taken from the name of the first routine in the file, which in most cases is one of the principal files in the library:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo% cd test_lib</td>
<td>Change directory to test_lib</td>
</tr>
<tr>
<td>demo% ls</td>
<td>List files in current directory</td>
</tr>
<tr>
<td></td>
<td>total 14 2 dropx.f 2 evalx.f 2 markx.f</td>
</tr>
<tr>
<td></td>
<td>2 delte.f 2 etc.f 2 linkz.f 2 point.f</td>
</tr>
</tbody>
</table>

The lower-level “helper” routines are gathered together into the file etc.f. The other files can contain one or more subprograms.

First, compile each of the library source files, using the –c option, to generate the corresponding relocatable .o files:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo% f77 –c *.f</td>
<td>Compile source files</td>
</tr>
</tbody>
</table>
| delte.f: | delte:
| q_fixx: | etc.f: |
| dropx.f: | q_fill:
| dropx: | q_step:
| etc.f: | q_node:
| q_warn: | etc |
| demo% ls | List files in current directory |
|           | total 42 2 dropx.f 4 etc.o 2 linkz.f 4 markx.o |
|           | 2 delte.f 4 dropx.o 2 evalx.f 4 linkz.o 2 point.f |
|           | 4 delte.o 2 etc.f 4 evalx.o 2 markx.f 4 point.o |

Now, create the static library testlib.a using ar:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo% ar cr testlib.a *.o</td>
<td>Create static library</td>
</tr>
</tbody>
</table>

Chapter 4 Libraries 53
To use this library, either include the library file on the compilation command or use the \texttt{-l} and \texttt{-L} compilation options. The example uses the .a file directly:

```
demo% cat trylib.f
C    program to test testlib routines
     x=21.998
     call evalx(x)
     call point(x)
     print*, 'value ',x
end
demo% f77 -o trylib trylib.f test_lib/testlib.a
trylib.f:
MAIN:
demo%
```

Notice that the main program calls only two of the routines in the library. You can verify that the uncalled routines in the library were not loaded into the executable file by looking for them in the list of names in the executable displayed by \texttt{nm}:

```
demo% nm trylib | grep FUNC | grep point
 [146] |    70016| 152|FUNC |GLOB |0 |8 |point_
demo% nm trylib | grep FUNC | grep evalx
 [165] |    69848| 152|FUNC |GLOB |0 |8 |evalx_
demo% nm trylib | grep FUNC | grep delte
demo% nm trylib | grep FUNC | grep markx
demo% ..etc
```

In the preceding example, \texttt{grep} finds entries in the list of names only for those library routines that were actually called.

Another way to reference the library is through the \texttt{-llibrary} and \texttt{-Lpath} options. Here, the library's name would have to be changed to conform to the \texttt{libname.a} convention:

```
demo% mv test_lib/testlib.a test_lib/libtestlib.a
demo% f77 -o trylib trylib.f -Ltest_lib -ltestlib
trylib.f:
MAIN:
```
The \texttt{-llibrary} and \texttt{-Lpath} options are used with libraries installed in a commonly accessible directory on the system, like \texttt{/usr/local/lib}, so that other users can reference it. For example, if you left \texttt{libtestlib.a} in \texttt{/usr/local/lib}, other users could be informed to compile with the following command:

\begin{verbatim}
demo$ f77 -o myprog myprog.f -L/usr/local/lib -ltestlib
\end{verbatim}

### Replacement in a Static Library

It is not necessary to recompile an entire library if only a few elements need recompiling. The \texttt{-r} option of \texttt{ar} permits replacement of individual elements in a static library.

Example: Recompile and replace a single routine in a static library:

\begin{verbatim}
demo$ f77 -c point.f
demo$ ar r testlib.a point.o
demo$
\end{verbatim}

### Ordering Routines in a Static Library

To order the elements in a static library when it is being built by \texttt{ar}, use the commands \texttt{lorder(1)} and \texttt{tsort(1)}:

\begin{verbatim}
demo$ ar cr mylib.a 'lorder exg.o fofx.o diffz.o | tsort'
\end{verbatim}

### Creating Dynamic Libraries

Dynamic library files are built by the linker \texttt{ld} from precompiled object modules that can be bound into the executable file after execution begins.

Another feature of a dynamic library is that modules can be used by other executing programs in the system without duplicating modules in each program’s memory. For this reason, a dynamic library is also a \textit{shared} library.

A dynamic library offers the following features:

- The object modules are \textit{not} bound into the executable file by the linker during the compile-link sequence; such binding is deferred until runtime.
A shared library module is bound into system memory when the first running program references it. If any subsequent running program references it, that reference is mapped to this first copy.

Maintaining programs is easier with dynamic libraries. Installing an updated dynamic library on a system immediately affects all the applications that use it without requiring relinking of the executable.

Tradeoffs for Dynamic Libraries

Dynamic libraries introduce some additional tradeoff considerations:

- Smaller a.out file

  Deferring binding of the library routines until execution time means that the size of the executable file is less than the equivalent executable calling a static version of the library; the executable file does not contain the binaries for the library routines.

- Possibly smaller process memory utilization

  When several processes using the library are active simultaneously, only one copy of the memory resides in memory and is shared by all processes.

- Possibly increased overhead

  Additional processor time is needed to load and link-edit the library routines during runtime. Also, the library’s position-independent coding might execute more slowly than the relocatable coding in a static library.

- Possible overall system performance improvement

  Reduced memory utilization due to library sharing should result in better overall system performance (reduced I/O access time from memory swapping).

Performance profiles among programs vary greatly from one to another. It is not always possible to determine or estimate in advance the performance improvement (or degradation) between dynamic versus static libraries. However, if both forms of a needed library are available to you, it would be worthwhile to evaluate the performance of your program with each.

Position-Independent Code and -pic

*Position-independent code* (PIC) can be bound to any address in a program without requiring relocation by the link editor. Such code is inherently sharable between simultaneous processes. Thus, if you are building a dynamic, shared library, you must compile the component routines to be position-independent (by using compiler options -pic or -PIC).
In position-independent code, each reference to a global item is compiled as a reference through a pointer into a global offset table. Each function call is compiled in a relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8 Kbytes on SPARC processors. The `-PIC` compiler option is similar to `-pic`, but `-PIC` allows the global offset table to span the range of 32-bit addresses.

There is a more flexible compiler flag with `f77` and `f95`, `-xcode=`, for specifying the code address space of a binary object. With this compiler flag, 32-, 44-, or 64-bit absolute addresses can be generated, as well as small and large model position-independent code. `-xcode=pic13` is equivalent to `-pic`, and `-xcode=pic32` is equivalent to `-PIC`. See the `f77(1)` and `f95(1)` man pages, or the `Fortran User's Guide`, for details.

**Binding Options**

You can specify dynamic or static library binding when you compile. These options are actually linker options, but they are recognized by the compiler and passed on to the linker.

```
-Bdynamic | -Bstatic
```

-Bdynamic sets the preference for shared, dynamic binding whenever possible.
-Bstatic restricts binding to static libraries only.

When both static and dynamic versions of a library are available, use this option to toggle between preferences on the command line:

```
f77 prog.f -Bdynamic -lwells -Bstatic -lsurface
```

```
-dy | -dn
```

Allows or disallows dynamic linking for the entire executable. (This option may appear on the command line only once.)

-dy allows dynamic, shared libraries to be linked. -dn does not allow linking of dynamic libraries.
Binding in 64-Bit Environments

Some static system libraries, such as libm.a and libc.a, are not available on 64-bit Solaris operating environments. These are supplied as dynamic libraries only. Use of -dn in these environments will result in an error indicating that some static system libraries are missing. Also, ending the compiler command line with -Bstatic will have the same effect.

To link with static versions of specific libraries, use a command line that looks something like:

```
f77 -o prog prog.f -Bstatic -labc -lxyz -Bdynamic
```

Here the user’s libabc.a and libxyz.a file are linked (rather than libabc.so or libxyz.so), and the final -Bdynamic insures that the remaining libraries, including system libraries, and dynamically linked.

In more complicated situations, it may be necessary to explicitly reference each system and user library on the link step with the appropriate -Bstatic or -Bdynamic as required. First use LD_OPTIONS set to `-Dfiles' to obtain a listing of all the libraries needed. Then perform the link step with -nolib (to suppress automatic linking of system libraries) and explicit references to the libraries you need. For example:

```
f77 -xarch=v9 -o cdf -nolib cdf.o-Bstatic -lF77 -lM77
    -lsunmath -Bdynamic -lm -lc
```

Naming Conventions

To conform to the dynamic library naming conventions assumed by the link loader and the compilers, assign names to the dynamic libraries that you create with the prefix lib and the suffix .so. For example, libmyfavs.so could be referenced by the compiler option -lmyfavs.

The linker also accepts an optional version number suffix: for example, libmyfavs.so.1 for version one of the library.

The compiler’s `-hname` option records `name` as the name of the dynamic library being built.

A Simple Dynamic Library

Building a dynamic library requires a compilation of the source files with the -pic or -PIC option and linker options -G, -ztext, and -hname. These linker options are available through the compiler command line.
You can create a dynamic library with the same files used in the static library example.

Example: Compile with –pic and other linker options:

```
.demo% f77 -o libtestlib.so.1 -G -pic -ztext -hlibtestlib.so.1 *.f
delte.f:
delte:
  q_fixx:
dropx.f:
dropx:
etc.f:
  q_fill:
  q_step:
  q_node:
  q_warn:
evalx.f:
evalx:
linkz.f:
linkz:
markx.f:
markx:
point.f:
  point:
Linking:
```

-G tells the linker to build a dynamic library.

-ztext warns you if it finds anything other than position-independent code, such as relocatable text.

Example: Make an executable file a.out using the dynamic library:

```
.demo% f77 -o trylib -R'pwd' trylib.f libtestlib.so.1
trylib.f:
MAIN main:
demo% file trylib
trylib:ELF 32-bit MSB executable SPARC Version 1, dynamically linked, not stripped
demo% ldd trylib
libtestlib.so.1 => /export/home/U/Tests/libtestlib.so.1
libF77.so.4 => /opt/SUNWspro/lib/libF77.so.4
libc.so.1 => /usr/lib/libc.so.1
libdl.so.1 => /usr/lib/libdl.so.1
```

Chapter 4 Libraries 59
Note that the example uses the \texttt{-R} option to bind into the executable the path (the current directory) to the dynamic library.

The \texttt{file} command shows that the executable is dynamically linked.

The \texttt{ldd} command shows that the executable, \texttt{trylib}, uses some shared libraries, including \texttt{libtestlib.so.1}; \texttt{libf77}, \texttt{libdl}, and \texttt{libc} are included by default by \texttt{f77}.

---

## Libraries Provided with Sun Fortran Compilers

The table shows the libraries installed with the compilers.

<table>
<thead>
<tr>
<th>Library</th>
<th>Name</th>
<th>Options Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{f77} functions, nonmath</td>
<td>\texttt{libF77}</td>
<td>None</td>
</tr>
<tr>
<td>\texttt{f77} functions, nonmath, multithread safe</td>
<td>\texttt{libF77_mt}</td>
<td>\texttt{-parallel}</td>
</tr>
<tr>
<td>\texttt{f77} math library</td>
<td>\texttt{libM77}</td>
<td>None</td>
</tr>
<tr>
<td>\texttt{f95} support intrinsics</td>
<td>\texttt{libfsu}</td>
<td>None</td>
</tr>
<tr>
<td>\texttt{f95} interface</td>
<td>\texttt{libfui}</td>
<td>None</td>
</tr>
<tr>
<td>\texttt{f95} array intrinsics libraries</td>
<td>\texttt{libf*ai}</td>
<td>None</td>
</tr>
<tr>
<td>\texttt{f95/\texttt{f77}} I/O compatibility library</td>
<td>\texttt{libf77compat}</td>
<td>\texttt{-lf77compat}</td>
</tr>
<tr>
<td>VMS library</td>
<td>\texttt{libV77}</td>
<td>\texttt{-lV77}</td>
</tr>
<tr>
<td>Library used with Pascal, Fortran, and C</td>
<td>\texttt{libpfc}</td>
<td>None</td>
</tr>
<tr>
<td>Library of Sun math functions</td>
<td>\texttt{libsunmath}</td>
<td>None</td>
</tr>
<tr>
<td>POSIX bindings</td>
<td>\texttt{libFposix}</td>
<td>\texttt{-lFposix}</td>
</tr>
<tr>
<td>\texttt{f95} POSIX interface</td>
<td>\texttt{libposix9}</td>
<td>\texttt{-lposix9}</td>
</tr>
<tr>
<td>POSIX bindings for extra runtime checking</td>
<td>\texttt{libFposix_c}</td>
<td>\texttt{-lFposix_c}</td>
</tr>
</tbody>
</table>

See also the \texttt{math\_libraries} README file for more information.
VMS Library

The libV77 library is the VMS library, which contains two special VMS routines, idate and time.

To use either of these routines, include the -lV77 option.

For idate and time, there is a conflict between the VMS version and the version that traditionally is available in UNIX environments. If you use the -lV77 option, you get the VMS compatible versions of the idate and time routines.

See the Fortran Library Reference Manual and the FORTRAN 77 Language Reference Manual for details on these routines.

POSIX Library

There are two versions of POSIX bindings provided with Fortran 77:

- libFposix, which is just the bindings (-lFposix)
- libFposix_c, which does some runtime checking to make sure you are passing correct handles (-lFposix_c)

If you pass bad handles:

- libFposix_c returns an error code (ENOHANDLE).
- libFposix core dumps with a segmentation fault.

Of course, the checking is time-consuming, and libFposix_c is several times slower.

Both POSIX libraries come in static and dynamic forms.

The POSIX bindings provided are for IEEE Standard 1003.9–1992.

IEEE 1003.9 is a binding of 1003.1–1990 to FORTRAN (X3.8–1978).

For more information, see these POSIX.1 documents:

- ISO/IEC 9945-1:1990
- IEEE Standard 1003.1–1990
- IEEE Order number SH13680
- IEEE CS Catalog number 1019

To find out precisely what POSIX is, you need both the 1003.9 and the POSIX.1 documents.

The POSIX library for f95 is libposix9.
Shippable Libraries

If your executable uses a Sun dynamic library that is listed in the runtime.libraries README file, your license includes the right to redistribute the library to your customer.

This README file is located in the READMEs directory:

<install-point>/SUNWspro/READMEs/

Do not redistribute or otherwise disclose the header files, source code, object modules, or static libraries of object modules in any form.

Refer to your software license for more details.
Program Analysis and Debugging

This chapter presents a number of Sun Fortran compiler features that facilitate program analysis and debugging.

Global Program Checking (\texttt{-Xlist})

The \texttt{-Xlist} options provide a valuable way to analyze a source program for inconsistencies and possible runtime problems. The analysis performed by the compiler is \textit{global}, across subprograms.

\texttt{-Xlist} reports errors in alignment, agreement in number and type for subprogram arguments, common block, parameter, and various other kinds of errors.

\texttt{-Xlist} also can be used to make detailed source code listings and cross-reference tables.

\textbf{Note} – Not all the \texttt{-Xlist} suboptions are available with \texttt{f95}.

GPC Overview

Global program checking (GPC), invoked by the \texttt{-Xlist} option, does the following:

- Enforces type-checking rules of Fortran more stringently than usual, especially between separately compiled routines
- Enforces some portability restrictions needed to move programs between different machines or operating systems
- Detects legal constructions that nevertheless might be suboptimal or error-prone
Reveals other potential bugs and obscurities

In particular, global checking reports problems such as:

- Interface problems
  - Conflicts in number and type of dummy and actual arguments
  - Wrong types of function values
  - Possible conflicts due to data type mismatches in common blocks between 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

How to Invoke Global Program Checking

The `-Xlist` option on the command line invokes the compiler’s global program analyzer. There are a number of suboptions, as described in the sections that follow.

Example: Compile three files for basic global program checking:

```bash
demo% f95 -Xlist any1.f any2.f any3.f
```

In the preceding example, the compiler:

- Produces output listings in the file `any1.lst`
- Compiles and links the program if there are no errors

Screen Output

Normally, output listings produced by `-Xlist` are written to a file. To display directly to the screen, use `-Xlisto` to write the output file to `/dev/tty`. 
Example: Display to terminal:

```
demo% f77 -Xlisto /dev/tty any1.f
```

Default Output Features

The `-Xlist` option provides a combination of features available for output. With no other `-Xlist` options, you get the following by default:

- The listing file name is taken from the first input source or object file that appears, with the extension replaced by `.lst`
- A line-numbered source listing
- Error messages (embedded in listing) for inconsistencies across routines
- Cross-reference table of the identifiers
- Pagination at 66 lines per page and 79 columns per line
- No call graph
- No expansion of `include` files

File Types

The checking process recognizes all the files in the compiler command line that end in `.f, .f90, .f95, .for, .F, .F95, or .o`. The `.o` files supply the process with information regarding only global names, such as subroutine and function names.

Analysis Files (.fln Files)

Programs compiled with `-Xlist` options have their analysis data built into the binary files automatically. This enables global program checking over programs in libraries.

Alternatively, the compiler will save individual source file analysis results into files with a `.fln` suffix if the `-Xlistfln` option is also specified. `dir` indicates the directory to receive these files.

```
demo% f77 -Xlistfln/tmp *.f
```
Some Examples of `-Xlist` and Global Program Checking

Here is a listing of the `Repeat.f` source code used in the following examples:

```fortran
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
```
Example: Use –XlistE to show errors and warnings:

```
demo% f77 -XlistE -silent Repeat.f

demo% cat Repeat.lst
FILE  "Repeat.f"
program  repeat
  4             CALL nwfrk ( pn1 )

  **** ERR  #418:  argument "pn1" is real, but dummy argument is integer*4
  See: "Repeat.f" line #14
  4             CALL nwfrk ( pn1 )

  **** ERR  #317: variable "pn1" referenced as integer*4 across repeat/nwfrk/prnok in line #21 but set as real by repeat in line #2

subroutine  subr1
  10              CALL subr1 ( x * 0.5 )

  **** WAR  #348: recursive call for "subr1". See dynamic calls: "Repeat.f" line #3

subroutine  nwfrk
  17             PRINT *, prnok ( ix ), fork ( )

  **** ERR  #418: argument "ix" is integer*4, but dummy argument is real
  See: "Repeat.f" line #20

subroutine  unreach_sub
  24           SUBROUTINE unreach_sub()

  **** WAR  #338: subroutine "unreach_sub" isn't called from program

Date:     Wed Feb 24 10:40:32 1999
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
```
Compiling the same program with `-Xlist` also produces a cross-reference table on standard output:

```
<table>
<thead>
<tr>
<th>CROSS REFERENCE TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source file: Repeat.f</td>
</tr>
<tr>
<td>Legend:</td>
</tr>
<tr>
<td>D Definition/Declaration</td>
</tr>
<tr>
<td>U Simple use</td>
</tr>
<tr>
<td>M Modified occurrence</td>
</tr>
<tr>
<td>A Actual argument</td>
</tr>
<tr>
<td>C Subroutine/Function call</td>
</tr>
<tr>
<td>I Initialization: DATA or extended declaration</td>
</tr>
<tr>
<td>E Occurrence in EQUIVALENCE</td>
</tr>
<tr>
<td>N Occurrence in NAMELIST</td>
</tr>
<tr>
<td>Program Form</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>&lt;repeat&gt; D 1:D</td>
</tr>
<tr>
<td>Functions and Subroutines</td>
</tr>
<tr>
<td>fork</td>
</tr>
<tr>
<td>int*4 &lt;nwfrk&gt; DC 15:D 16:D 17:C</td>
</tr>
<tr>
<td>int intrinsic</td>
</tr>
<tr>
<td>&lt;prnok&gt; C 21:C</td>
</tr>
<tr>
<td>loc intrinsic</td>
</tr>
<tr>
<td>&lt;repeat&gt; C 2:C</td>
</tr>
<tr>
<td>&lt;prnok&gt; C 21:C</td>
</tr>
<tr>
<td>nwfrk</td>
</tr>
<tr>
<td>&lt;repeat&gt; C 4:C</td>
</tr>
<tr>
<td>&lt;nwfrk&gt; D 14:D</td>
</tr>
<tr>
<td>prnok int*4 &lt;nwfrk&gt; DC 16:D 17:C</td>
</tr>
<tr>
<td>&lt;prnok&gt; DM 20:D 21:M</td>
</tr>
<tr>
<td>real intrinsic</td>
</tr>
<tr>
<td>&lt;repeat&gt; C 2:C</td>
</tr>
<tr>
<td>sleep</td>
</tr>
<tr>
<td>&lt;unreach_sub&gt; C 25:C</td>
</tr>
<tr>
<td>subr1</td>
</tr>
<tr>
<td>&lt;repeat&gt; C 3:C</td>
</tr>
<tr>
<td>&lt;subr1&gt; DC 8:D 10:C</td>
</tr>
<tr>
<td>unreach_sub</td>
</tr>
<tr>
<td>&lt;unreach_sub&gt; D 24:D</td>
</tr>
</tbody>
</table>
```
In the cross-reference table in the preceding example:

- **ix** is a 4-byte integer:
  - Used as an argument in the routine *nwfrk*
  - At line 14, used as a declaration of argument
  - At line 17, used as an actual argument

- **pn1** is a 4-byte real in the routine *repeat*:
  - At line 2, modified
  - At line 3, argument
  - At line 4, argument
  - At line 5, used

- **rp1** is a 4-byte real in the routine, *repeat*. At line 2, it is an argument.

- **x** is a 4-byte real in the routines *subr1* and *prnok*:
  - In *subr1*, at line 8, defined; used at lines 9 and 10
  - In *prnok*, at line 20, defined; at line 21, used as an argument
Suboptions for Global Checking Across Routines

The basic global cross-checking option is –Xlist with no suboption. It is a combination of suboptions, each of which could have been specified separately.

The following sections describe options for producing the listing, errors, and cross-reference table. Multiple suboptions may appear on 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.
- Use only one suboption per –Xlist.

–Xlist and its Suboptions

Combine suboptions according to the following rules:
- The most general option is –Xlist (listing, errors, cross-reference table).
- Specific features can be combined using –Xlistc, –XlistE, –XlistL, or –XlistX.
- Other suboptions specify further details.

Example: Each of these two command lines performs the same task:

```
demo% f77 –Xlistc –Xlist any.f
```
```
demo% f77 –Xlistc any.f
```

The following table shows the reports generated by these basic –Xlist suboptions alone:

<table>
<thead>
<tr>
<th>TABLE 5-1 Xlist Suboptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated Report</td>
</tr>
<tr>
<td>Errors, listing, cross-reference</td>
</tr>
<tr>
<td>Errors only</td>
</tr>
</tbody>
</table>
The following table summarizes all -Xlist suboptions.

### TABLE 5-2 Summary of -Xlist Suboptions

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xlist (no suboption)</td>
<td>Shows errors, listing, and cross-reference table</td>
</tr>
<tr>
<td>-Xlistc</td>
<td>Shows call graphs and errors (f77 only)</td>
</tr>
<tr>
<td>-XlistE</td>
<td>Shows errors</td>
</tr>
<tr>
<td>-Xlisterr[nnn]</td>
<td>Suppresses error nnn in the verification report</td>
</tr>
<tr>
<td>-Xlistf</td>
<td>Produces fast output</td>
</tr>
<tr>
<td>-Xlistflndir</td>
<td>Puts the .fln files in dir (f77 only)</td>
</tr>
<tr>
<td>-Xlisth</td>
<td>Shows errors from cross-checking stop compilation (f77 only)</td>
</tr>
<tr>
<td>-XlistI</td>
<td>Lists and cross-checks include files</td>
</tr>
<tr>
<td>-XlistL</td>
<td>Shows the listing and errors</td>
</tr>
<tr>
<td>-Xlistln</td>
<td>Sets page breaks</td>
</tr>
<tr>
<td>-Xlisto name</td>
<td>Renames the -Xlist output report file</td>
</tr>
<tr>
<td>-Xlists</td>
<td>Suppresses unreferenced symbols from cross-reference (f77 only)</td>
</tr>
<tr>
<td>-Xlistvn</td>
<td>Sets checking “strictness” level (f77 only)</td>
</tr>
<tr>
<td>-Xlistw[nnn]</td>
<td>Sets the width of output lines (f77 only)</td>
</tr>
<tr>
<td>-Xlistwar[nnn]</td>
<td>Suppresses warning nnn in the report</td>
</tr>
<tr>
<td>-XlistX</td>
<td>Shows just the cross-reference table and errors</td>
</tr>
</tbody>
</table>

-Xlist Suboption Reference

This section describes the -Xlist suboptions. As noted, some are only available with f77.
**f77: -Xlistc — Show call graphs and cross-routine errors**

Used alone, -Xlistc does not show a listing or cross-reference. It produces the call graph in a tree form, using printable characters. If some subroutines are not called from MAIN, more than one graph is shown. Each BLOCKDATA is printed separately with no connection to MAIN.

The default is *not* to show the call graph.

**-XlistE — Show cross-routine errors**

Used alone, -XlistE shows only cross-routine errors and does not show a listing or a cross-reference.

**-Xlisterr[nnn] — Suppress error nnn**

Use -Xlisterr to suppress a numbered error message from the listing or cross-reference.

For example: -Xlisterr338 suppresses error message 338. If nnn is not specified, all error messages are suppressed. To suppress additional specific errors, use this option repeatedly.

**-Xlistf — Produce faster output**

Use -Xlistf to produce source file listings and a cross-checking report and to verify sources, but without generating object files.

The default without this option is to generate object files.

**f77: -Xlistfln dir — Put .fln files into dir directory**

Use -Xlistfln to specify the directory to receive .fln source analysis files. The directory specified (dir) must already exist. The default is to include the source analysis information directly within the object .o files (and not generate .fln files).

**f77: -Xlisth — Halt on errors**

With -Xlisth, compilation stops if errors are detected while cross-checking the program. In this case, the report is redirected to stdout instead of the *.lst file.
-XlistI – List and cross-check include files

If -XlistI is the only suboption used, include files are shown or scanned along with the standard -Xlist output (line numbered listing, error messages, and a cross-reference table).

- 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 files are:
  - Source files
  - #include files
  - INCLUDE files

- Cross-Reference Table—If the cross reference table is not suppressed, the following files are all scanned while the cross reference table is generated:
  - Source files
  - #include files
  - INCLUDE files

The default is not to show include files.

- XlistL – Show listing and cross routine errors

Use -XlistL to produce only a listing and a list of cross routine errors. This suboption by itself does not show a cross reference table. The default is to show the listing and cross reference table.

- XlistLn – Set the page length for pagination to n lines

Use -Xlistl to set the page length to something other than the default page size. For example, -Xlist145 sets the page length to 45 lines. The default is 66.

With n=0 (-Xlist0) this option shows listings and cross-references with no page breaks for easier on-screen viewing.

- Xlisto name – Rename the -Xlist output report file

Use -Xlisto to rename the generated report output file. (A space between o and name is required.) With -Xlisto name, the output is to name.lst.

To display directly to the screen, use the command: -Xlisto /dev/tty
**f77: -Xlists – Suppress unreferenced identifiers**

Use `-Xlists` to suppress from the cross reference table any identifiers defined in the `include` files but not referenced in the source files.

This suboption has no effect if the suboption `-XlistI` is used.

The default is not to show the occurrences in `#include` or `INCLUDE` files.

**f77: -Xlistvn – Set level of checking strictness**

`n` is 1, 2, 3, or 4. The default is 2 (-Xlistv2):

- `-Xlistv1`
  Shows the cross-checked information of all names in summary form only, with no line numbers. This is the lowest level of checking strictness—syntax errors only.

- `-Xlistv2`
  Shows cross-checked information with summaries and line numbers. This is the default level of checking strictness and includes argument inconsistency errors and variable usage errors.

- `-Xlistv3`
  Shows cross-checking with summaries, line numbers, and common block maps. This is a high level of checking strictness and includes errors caused by incorrect usage of data types in common blocks in different subprograms.

- `-Xlistv4`
  Shows cross-checking with summaries, line numbers, common block maps, and equivalence block maps. This is the strictest level of checking with maximum error detection.

**f77: -Xlistw[nnn] – Set width of output line to `n` columns**

Use `-Xlistw` to set the width of the output line. For example, `-Xlistw132` sets the page width to 132 columns. The default is 79.

**-Xlistwar[nnn] – Suppress warning `nnn` in the report**

Use `-Xlistwar` to suppress a specific warning message from the output reports. If `nnn` is not specified, then all warning messages are suppressed from printing. For example, `-Xlistwar338` suppresses warning message number 338. To suppress more than one, but not all warnings, use this option repeatedly.
–XlistX – Show cross-reference table and cross routine errors

–XlistX produces a cross reference table and cross routine error list but no source listing.

Some Examples Using Suboptions

Example: Use –Xlistwarnnn to suppress two warnings from a preceding example:

demo% f77 -Xlistwar338 -Xlistwar348 -XlistE -silent Repeat.f
demo% cat Repeat.lst
FILE  "Repeat.f"
program  repeat
  4    CALL nwfrk ( pn1 )
        ^
    **** ERR #418:  argument "pn1" is real, but dummy argument is integer*4
    See: "Repeat.f" line #14
    4    CALL nwfrk ( pn1 )
        ^
    **** ERR #317:  variable "pn1" referenced as integer*4 across repeat/nwfrk/prnok in line #21 but set as real by repeat in line #2
subroutine  nwfrk
  17       PRINT *, prnok ( ix ), fork ( )
        ^
    **** ERR #418:  argument "ix" is integer*4, but dummy argument is real
    See: "Repeat.f" line #20

Date:     Wed Feb 24 10:40:32 1999
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo%
Example: Explain a message and find a type mismatch in program ShoGetc.f:

```fortran
CHARACTER*1 c
i = getc(c)
END
```

```bash
demo% cat ShoGetc.f
CHARACTER*1 c
i = getc(c)
END
demo% f77 -silent ShoGetc.f
Compile program
demo% a.out
Program waits for input...
Z
Type "Z" on keyboard. This causes run-time message. Why?
```

Note: IEEE floating-point exception flags raised:
Invalid Operation;
See the Numerical Computation Guide, ieee_flags(3M)

```bash
demo% f77 -XlistE -silent ShoGetc.f
Compile with Global Program Checking and view listing
demo% cat ShoGetc.lst
FILE "ShoGetc.f"
program MAIN
2          i = getc(c)
^
**** WAR #320: variable "i" set but never referenced
2          i = getc(c)
^
**** ERR #412: function "getc" used as real but declared as integer*4
Here is the error - function must be declared INTEGER.
2          i = getc(c)
^
**** WAR #320: variable "c" set but never referenced
```

```bash
demo% cat ShoGetc.f
Modify program to declare getc INTEGER and run again.
CHARACTER*1 c
INTEGER getc
i = getc(c)
END
demo% f77 -silent ShoGetc.f
demo% a.out
Z
Type "Z" on keyboard
```

```bash
demo% Now no error.
```
Special Compiler Options

Some compiler options are useful for debugging. They check subscripts, spot undeclared variables, show stages of the compile-link sequence, display versions of software, and so on.

The Solaris linker has additional debugging aids. See `ld(1)`, or run the command `ld -Dhelp` at a shell prompt to see the online documentation.

Subscript Bounds (–C)

The –C option adds checks for out-of-bounds array subscripts.

If you compile with –C, the compiler adds checks at runtime for out-of-bounds references on each array subscript. This action helps catch some situations that cause segmentation faults.

Example: Index out of range:

```bash
demo$ cat indrange.f
REAL a(10,10)
k = 11
   a(k,2) = 1.0
END
demo$ f77 -C -silent indrange.f
demo$ a.out
```

Undeclared Variable Types (–u)

The –u option checks for any undeclared variables.

The –u option causes all variables to be initially identified as undeclared, so that all variables that are not explicitly declared by type statements, or by an IMPLICIT statement, are flagged with an error. The –u flag is useful for discovering mistyped variables. If –u is set, all variables are treated as undeclared until explicitly declared. Use of an undeclared variable is accompanied by an error message.
Version Checking (\texttt{–V})

The \texttt{–V} option causes the name and version ID of each phase of the compiler to be displayed. This option can be useful in tracking the origin of ambiguous error messages and in reporting compiler failures, and to verify the level of installed compiler patches.

Interactive Debugging With \texttt{dbx} and Sun WorkShop

The Sun WorkShop provides a tightly integrated development environment for building and browsing, as well as debugging applications written in Fortran, C, and C++.

The Sun WorkShop debugging facility is a window-based interface to \texttt{dbx}, while \texttt{dbx} itself is an interactive, line-oriented, source-level symbolic debugger. Either can be used to determine where a program crashed, to view or trace the values of variables and expressions in a running code, and to set breakpoints.

Sun WorkShop adds a sophisticated graphical environment to the debugging process that is integrated with tools for editing, building, and source code version control. It includes a data visualization capability to display and explore large and complex datasets, simulate results, and interactively steer computations.

For details, see the Sun manual \textit{Debugging a Program With Sun WorkShop}, and the \texttt{dbx(1)} man pages.

The \texttt{dbx} program provides event management, process control, and data inspection. You can watch what is happening during program execution, and perform the following tasks:

- Fix one routine, then continue executing without recompiling the others
- Set watchpoints to stop or trace if a specified item changes
- Collect data for performance tuning
- Graphically monitor variables, structures, and arrays
- 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 a program, one source or assembly line at a time
- 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
- Save and then replay all or part of a debugging run
■ Examine the call stack, or move up and down the call stack
■ Program scripts in the embedded Korn shell
■ Follow programs as they fork(2) and exec(2)

To debug optimized programs, use the dbx fix command to recompile the routines you want to debug:

1. **Compile the program with the appropriate −O n optimization level.**
2. **Start the execution under dbx.**
3. **Use fix −g any . f without optimization on the routine you want to debug.**
4. **Use continue with that routine compiled.**

Some optimizations will be inhibited by the presence of −g on the compilation command. See the dbx documentation for details.

---

**$77$: Viewing Compiler Listing Diagnostics**

Use the error utility program to view compiler diagnostics merged with the source code. error inserts compiler diagnostics above the relevant line in the source file. The diagnostics include the standard compiler error and warning messages, but not the −Xlist error and warning messages.

**Note** – The error utility rewrites your source files and does not work if the source files are read-only, or are in a read only directory.

error(1) is included as part of a “developer” installation of the Solaris operating environment; it can also be installed from the package, SUNWbtool.

Facilities also exist in the Sun WorkShop for viewing compiler diagnostics. See Introduction to Sun WorkShop.
Floating-Point Arithmetic

This chapter considers floating-point arithmetic and suggests strategies for avoiding and detecting numerical computation errors.

For a detailed examination of floating-point computation on SPARC and x86 processors, see the Sun Numerical Computation Guide.

Introduction

Sun’s floating-point environment on SPARC and x86 implements the arithmetic model specified by the IEEE Standard 754 for Binary Floating Point Arithmetic. This environment enables you to develop robust, high-performance, portable numerical applications. It also provides tools to investigate any unusual behavior by a numerical program.

In numerical programs, there are many potential sources for computational error:

- The computational model could be wrong.
- The algorithm used could be numerically unstable.
- The data could be ill-conditioned.
- The hardware could be producing unexpected results.

Finding the source of the errors in a numerical computation that has gone wrong can be extremely difficult. The chance of coding errors can be reduced by using commercially available and tested library packages whenever possible. Choice of algorithms is another critical issue. Using the appropriate computer arithmetic is another.

This chapter makes no attempt to teach or explain numerical error analysis. The material presented here is intended to introduce the IEEE floating-point model as implemented by Sun WorkShop Fortran compilers.
IEEE Floating-Point Arithmetic

IEEE arithmetic is a relatively new way of dealing with arithmetic operations that result in such problems as invalid, division by zero, overflow, underflow, or inexact. The differences are in rounding, handling numbers near zero, and handling numbers near the machine maximum.

The IEEE standard supports user handling of exceptions, rounding, and precision. Consequently, the standard supports interval arithmetic and diagnosis of anomalies. IEEE Standard 754 makes it possible to standardize elementary functions like \( \exp \) and \( \cos \), to create high precision arithmetic, and to couple numerical and symbolic algebraic computation.

IEEE arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic. The standard simplifies the task of writing numerically sophisticated, portable programs. Many questions about floating-point arithmetic concern elementary operations on numbers. For example:

- What is the result of an operation when the infinitely precise result is not representable in the computer hardware?
- Are elementary operations like multiplication and addition commutative?

Another class of questions concerns floating-point exceptions and exception handling. What happens if you:

- Multiply two very large numbers with the same sign?
- Divide nonzero by zero?
- Divide zero by zero?

In older arithmetic models, the first class of questions might not have the expected answers, while the exceptional cases in the second class might all have the same result: the program aborts on the spot or proceeds with garbage results.

The standard ensures that operations yield the mathematically expected results with the expected properties. It also ensures that exceptional cases yield specified results, unless the user specifically makes other choices.

For example, the exceptional values +Inf, -Inf, and NaN are introduced intuitively:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>big*big</td>
<td>+Inf</td>
</tr>
<tr>
<td>big*(-big)</td>
<td>-Inf</td>
</tr>
<tr>
<td>num/0.0</td>
<td>+Inf</td>
</tr>
<tr>
<td>num/0.0</td>
<td>-Inf</td>
</tr>
<tr>
<td>0.0/0.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Also, five types of floating-point exception are identified:

- **Invalid.** Operations with mathematically invalid operands—for example, 0.0/0.0, \( \sqrt{-1.0} \), and \( \log(-37.8) \)
- **Division by zero.** Divisor is zero and dividend is a finite nonzero number—for example, 9.9/0.0
- **Overflow.** Operation produces a result that exceeds the range of the exponent—for example, MAXDOUBLE+0.0000000000001e308
- **Underflow.** Operation produces a result that is too small to be represented as a normal number—for example, MINDOUBLE * MINDOUBLE
- **Inexact.** Operation produces a result that cannot be represented with infinite precision—for example, 2.0 / 3.0, \( \log(1.1) \) and 0.1 in input

The implementation of the IEEE standard is described in the Sun *Numerical Computation Guide*.

### –ftrap=mode Compiler Options

The –ftrap=mode option enables trapping for floating-point exceptions. If no signal handler has been established by an `ieee_handler()` call, the exception terminates the program with a memory dump core file. See *Fortran User's Guide* for details on this compiler option. For example, to enable trapping for overflow, division by zero, and invalid operations, compile with –ftrap=common.

**Note** – You must compile the application’s main program with –ftrap= for trapping to be enabled.

### Floating-Point Exceptions and Fortran

Programs compiled by *f77* automatically display a list of accrued floating-point exceptions on program termination. In general, a message results if any one of the invalid, division-by-zero, or overflow exceptions have occurred. Inexact exceptions do not generate messages because they occur so frequently in real programs.

*f95* programs do not automatically report on exceptions at program termination. An explicit call to `ieee_retrospective(3M)` is required.

You can turn off any or all of these messages with `ieee_flags()` by clearing exception status flags. Do this at the end of your program.
Handling Exceptions

Exception handling according to the IEEE standard is the default on SPARC and x86 processors. However, there is a difference between detecting a floating-point exception and generating a signal for a floating-point exception (SIGFPE).

Following the IEEE standard, two things happen when an untrapped exception occurs during a floating-point operation:
- The system returns a default result. For example, on 0/0 (invalid), the system returns NaN as the result.
- A flag is set to indicate that an exception is raised. For example, 0/0 (invalid), the system sets the “invalid operation” flag.

Trapping a Floating-Point Exception

$\texttt{f77}$ and $\texttt{f95}$ differ significantly in the way they handle floating-point exceptions.

With $\texttt{f77}$, the default on SPARC and x86 systems is not to automatically generate a signal to interrupt the running program for a floating-point exception. The assumptions are that signals could degrade performance and that most exceptions are not significant as long as expected values are returned.

The default with $\texttt{f95}$ is to automatically trap on division by zero, overflow, and invalid operation.

The $\texttt{f77}$ and $\texttt{f95}$ command-line option $\texttt{-ftrap}$ can be used to change the default. In terms of $\texttt{-ftrap}$, the default for $\texttt{f77}$ is $\texttt{-ftrap=none}$. The default for $\texttt{f95}$ is $\texttt{-ftrap=common}$.

To enable exception trapping, compile the main program with one of the $\texttt{-ftrap}$ options—for example: $\texttt{-ftrap=common}$.

SPARC: Nonstandard Arithmetic

One aspect of standard IEEE arithmetic, called gradual underflow, can be manually disabled. When disabled, the program is considered to be running with nonstandard arithmetic.

The IEEE standard for arithmetic specifies a way of handling underflowed results gradually by dynamically adjusting the radix point of the significand. 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 shifts the radix point into the significand when the result of a floating-point computation would otherwise underflow. With a SPARC processor
this result is not accomplished in hardware but in software. If your program generates many underflows (perhaps a sign of a problem with your algorithm) and you run on a SPARC processor, you may experience a performance loss.

Gradual underflow can be disabled either by compiling with the –fns option or by calling the library routine nonstandard_arithmetic() from within the program to turn it off. Call standard_arithmetic() to turn gradual underflow back on.

**Note** – To be effective, the application’s main program must be compiled with –fns. See the *Fortran User’s Guide*.

For legacy applications, take note that:
- The `standard_arithmetic()` subroutine replaces an earlier routine named `gradual_underflow()`.
- The `nonstandard_arithmetic()` subroutine replaces an earlier routine named `abrupt_underflow()`.

**Note** – The –fns option and the `nonstandard_arithmetic()` library routine are effective only on some SPARC systems. On x86 platforms, gradual underflow is performed by the hardware.

### IEEE Routines

The following interfaces help people use IEEE arithmetic and are described in man pages. These are mostly in the math library `libsunmath` and in several `.h` files.
- `ieee_flags(3m)`—Controls rounding direction and rounding precision; query exception status; clear exception status
- `ieee_handler(3m)`—Establishes an exception handler routine
- `ieee_functions(3m)`—Lists name and purpose of each IEEE function
- `ieee_values(3m)`—Lists functions that return special values
- Other *libm* functions described in this section:
  - `ieee_retrospective`
  - `nonstandard_arithmetic`
  - `standard_arithmetic`

The SPARC processors conform to the IEEE standard in a combination of hardware and software support for different aspects. x86 processors conform to the IEEE standard entirely through hardware support.
The newest SPARC processors contain floating-point units with integer multiply and divide instructions and hardware square root.

Best performance is obtained when the compiled code properly matches the runtime floating-point hardware. The compiler’s \texttt{-xtarget=} option permits specification of the runtime hardware. For example, \texttt{-xtarget=ultra} would inform the compiler to generate object code that will perform best on an UltraSPARC processor.

\textit{On SPARC platforms}: The utility \texttt{fpversion} displays which floating-point hardware is installed and indicates the appropriate \texttt{-xtarget} value to specify. This utility runs on all Sun SPARC architectures. See \texttt{fpversion(1)}, the Sun WorkShop \textit{Fortran User’s Guide} (regarding \texttt{-xtarget}) and the \textit{Numerical Computation Guide} for details.

\textbf{Flags and \texttt{ieee\_flags()}}

The \texttt{ieee\_flags} function is used to query and clear exception status flags. It is part of the \texttt{libsunmath} library shipped with Sun compilers and performs the following tasks:

- Controls rounding direction and rounding precision
- Checks the status of the exception flags
- Clears exception status flags

The general form of a call to \texttt{ieee\_flags} is:

\begin{verbatim}
flags = ieee_flags( action, mode, in, out )
\end{verbatim}

Each of the four arguments is a string. The input is \texttt{action}, \texttt{mode}, and \texttt{in}. The output is \texttt{out} and \texttt{flags}. \texttt{ieee\_flags} is an integer-valued function. Useful information is returned in \texttt{flags} as a set of 1-bit flags. Refer to the man page for \texttt{ieee\_flags(3m)} for complete details.
Possible parameter values are shown in the following table:

**TABLE 6-1**  \( \text{ieee\_flags}( \text{action}, \text{mode}, \text{in}, \text{out} ) \) Argument Values

<table>
<thead>
<tr>
<th>action</th>
<th>mode</th>
<th>in, out</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>direction</td>
<td>nearest</td>
</tr>
<tr>
<td>set</td>
<td>precision</td>
<td>tozero</td>
</tr>
<tr>
<td>clear</td>
<td>exception</td>
<td>negative</td>
</tr>
<tr>
<td>clearall</td>
<td>exception</td>
<td>positive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>single</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inexact</td>
</tr>
<tr>
<td></td>
<td></td>
<td>division</td>
</tr>
<tr>
<td></td>
<td></td>
<td>underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>common</td>
</tr>
</tbody>
</table>

The precision mode is available only on x86 platforms.

Note that these are literal character strings, and the output parameter out must be at least CHARACTER*9. 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 6-2**  \( \text{ieee\_flags} \) Argument Meanings

<table>
<thead>
<tr>
<th>Value of in and out</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>nearest, tozero, negative, positive</td>
<td>Rounding direction</td>
</tr>
<tr>
<td>extended, double, single</td>
<td>Rounding precision</td>
</tr>
<tr>
<td>inexact, division, underflow, overflow, invalid</td>
<td>Exceptions</td>
</tr>
<tr>
<td>all</td>
<td>All five exceptions</td>
</tr>
<tr>
<td>common</td>
<td>Common exceptions: invalid, division, overflow</td>
</tr>
</tbody>
</table>
For example, to determine what is the highest priority exception that has a flag raised, pass the input argument *in* as the null string:

```fortran
CHARACTER *9, out
ieeer = ieee_flags( 'get', 'exception', '', out )
PRINT *, out, ' flag raised'
```

Also, 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.

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

Example: Clear the *invalid* exception:

```fortran
ieeer = ieee_flags( 'clear', 'exception', 'invalid', out )
```

Example: Clear all exceptions:

```fortran
ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

Example: Set rounding direction to zero:

```fortran
ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

Example: Set rounding precision to *double*:

```fortran
ieeer = ieee_flags( 'set', 'precision', 'double', out )
```

**Turning Off All Warning Messages With *ieee_flags***

Calling *ieee_flags* with an *action* of *clear*, as shown in the following example, resets any uncleared exceptions. Put this call before the program exits to suppress system warning messages about floating-point exceptions at program termination.
Example: Clear all accrued exceptions with `ieee_flags()`: 

```fortran
i = ieee_flags('clear', 'exception', 'all', out)
```

### Detecting an Exception With `ieee_flags`

The following example demonstrates how to determine which floating-point exceptions have been raised by earlier computations. Bit masks defined in the system include file `floatingpoint.h` are applied to the value returned by `ieee_flags`.

**Note** – Fortran 95 (`f95`) programs should include the file `floatingpoint.h`; Fortran 77 (`f77`) programs should include `f77_floatingpoint.h`.

In this example, `DetExcFlg.F`, the include file is introduced using the `#include` preprocessor directive, which requires us to name the source file with a `.F` suffix. Underflow is caused by dividing the smallest double-precision number by 2.

Example: Detect an exception using `ieee_flags` and decode it:

```fortran
#include "floatingpoint.h"

CHARACTER*16 out

DOUBLE PRECISION d_max_subnormal, x
INTEGER div, flgs, inv, inx, over, under

x = d_max_subnormal() / 2.0                ! Cause underflow
flgs=ieee_flags('get','exception','',out) ! Which are raised?
inx   = and(rshift(flgs, fp_inexact)  , 1)  ! Decode
div   = and(rshift(flgs, fp_division) , 1)   ! the value
under = and(rshift(flgs, fp_underflow), 1)    ! returned
over  = and(rshift(flgs, fp_overflow) , 1)     ! by
inv   = and(rshift(flgs, fp_invalid)  , 1)      ! `ieee_flags`

PRINT *, "Highest priority exception is: ", out
PRINT *, 1 invalid divide overflo underflo inexact'
PRINT '(5i8)', inv, div, over, under, inx
PRINT *, '(1 = exception is raised; 0 = it is not)'
i = ieee_flags('clear', 'exception', 'all', out) ! Clear all
END
```
Example: Compile and run the preceding example (DetExcFlg.F):

```fortran
demo% f95 DetExcFlg.F
demo% a.out
Highest priority exception is: underflow
  invalid  divide  overflo underflo inexact
  0       0       0       1       1
(l = exception is raised; 0 = it is not)
demo%
```

IEEE Extreme Value Functions

The compilers provide a set of functions that can be called to return a special IEEE extreme value. These values, such as infinity or minimum normal, can be used directly in an application program.

Example: A convergence test based on the smallest number supported by the hardware would look like:

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

The values available are listed in the following table:

<table>
<thead>
<tr>
<th>IEEE Value</th>
<th>Double Precision</th>
<th>Single Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinity</td>
<td>d_infinity()</td>
<td>r_infinity()</td>
</tr>
<tr>
<td>quiet NaN</td>
<td>d_quiet_nan()</td>
<td>r_quiet_nan()</td>
</tr>
<tr>
<td>signaling NaN</td>
<td>d_signaling_nan()</td>
<td>r_signaling_nan()</td>
</tr>
<tr>
<td>min normal</td>
<td>d_min_normal()</td>
<td>r_min_normal()</td>
</tr>
<tr>
<td>min subnormal</td>
<td>d_min_subnormal()</td>
<td>r_min_subnormal()</td>
</tr>
<tr>
<td>max subnormal</td>
<td>d_max_subnormal()</td>
<td>r_max_subnormal()</td>
</tr>
<tr>
<td>max normal</td>
<td>d_max_normal()</td>
<td>r_max_normal()</td>
</tr>
</tbody>
</table>

The two NaN values (quiet and signaling) are unordered and should not be used in comparisons such as IF(X.ne.r_quiet_nan()) THEN... To determine whether some value is a NaN, use the function ir_isnan(r) or id_isnan(d).
The Fortran names for these functions are listed in these man pages:

- `libm_double(3f)`
- `libm_single(3f)`
- `ieee_functions(3m)`

Also see:

- `ieee_values(3m)`
- The `floatingpoint.h` and `f77_floatingpoint.h` header files

**Exception Handlers and ieee_handler()**

Typical concerns about IEEE exceptions are:

- What happens when an exception occurs?
- How do I use `ieee_handler()` to establish a user function as an exception handler?
- How do I write a function that can be used as an exception handler?
- How do I locate the exception—where did it occur?

Exception trapping to a user routine begins with the system generating a signal on a floating-point exception. The standard UNIX name for signal: floating-point exception is `SIGFPE`. The default situation on SPARC and x86 platforms is not to generate a `SIGFPE` when an exception occurs. For the system to generate a `SIGFPE`, exception trapping must first be enabled, usually by a call to `ieee_handler()`.

**Establishing an Exception Handler Function**

To establish a function as an exception handler, pass the name of the function to `ieee_handler()`, together with the name of the exception to watch for and the action to take. Once you establish a handler, a `SIGFPE` signal is generated whenever the particular floating-point exception occurs, and the specified function is called.
The form for invoking `ieee_handler()` is shown in the following table:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>character</td>
<td>get, set, or clear</td>
</tr>
<tr>
<td>exception</td>
<td>character</td>
<td>invalid, division, overflow, underflow,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or inexact</td>
</tr>
<tr>
<td>handler</td>
<td>Function name</td>
<td>The name of the user handler function or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIGFPE_DEFAULT, SIGFPE.Ignore, orSIGFPE_ABORT</td>
</tr>
</tbody>
</table>

Return value | integer | 0 =OK |

A Fortran 77 routine compiled with `f77` that calls `ieee_handler()` should also declare:

```c
#include 'f77_floatingpoint.h'
```

For `f95` programs, declare:

```c
#include 'floatingpoint.h'
```

The special arguments `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, and `SIGFPE_ABORT` are defined in these include files and can be used to change the behavior of the program for a specific exception:

<table>
<thead>
<tr>
<th>Special Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGFPE_DEFAULT</td>
<td>No action taken when the specified exception occurs.</td>
</tr>
<tr>
<td>SIGFPE_IGNORE</td>
<td>Program aborts, possibly with dump file, on exception.</td>
</tr>
<tr>
<td>SIGFPE_ABORT</td>
<td>Program aborts, possibly with dump file, on exception.</td>
</tr>
</tbody>
</table>

### Writing User Exception Handler Functions

The actions your exception handler takes are up to you. However, the routine must be an integer function with three arguments specified as shown:

```fortran
handler_name( sig, sip, uap )
```

- `handler_name` is the name of the integer function.
- `sig` is an integer.
- `sip` is a record that has the structure `siginfo`.
- `uap` is not used.
Example: An exception handler function:

```
INTEGER FUNCTION hand( sig, sip, uap )
INTEGER sig, location
STRUCTURE /fault/
   INTEGER address
   INTEGER trapno
END STRUCTURE
STRUCTURE /siginfo/
   INTEGER si_signo
   INTEGER si_code
   INTEGER si_errno
   RECORD /fault/ fault
END STRUCTURE
RECORD /siginfo/ sip
location = sip.fault.address
... actions you take ...
END
```

This `f77` example would have to be modified to run on SPARC V9 architectures (`-xarch=v9` or `v9a`) by replacing all INTEGER declarations within each STRUCTURE with INTEGER*8.

If the handler routine enabled by `ieee_handler()` is in Fortran as shown in the example, the routine should not make any reference to its first argument (`sig`). This first argument is passed by value to the routine and can only be referenced as `loc(sig)`. The value is the signal number.

**Detecting an Exception by Handler**

The following examples show how to create handler routines to detect floating-point exceptions.
Example: Detect exception and abort:

```fortran
DEMO% cat DetExcHan.f
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)
INTEGER sig, code, context(5)
CALL abort()
END
DEMO% f77 -silent DetExcHan.f
DEMO% a.out
abort: called
Abort (core dumped)
DEMO%
```

SIGFPE is generated whenever that floating-point exception occurs. When the
SIGFPE is detected, control passes to the myhandler function, which immediately
aborts. Compile with -g and use dbx to find the location of the exception.

**Locating an Exception by Handler**

Example: Locate an exception (print address) and abort:
In SPARC V9 environments, replace the INTEGER declarations within each STRUCTURE with INTEGER*8, and the i4 formats with i8.
In most cases, knowing the actual *address* of the exception is of little use, except with dbx:

```
demo% dbx a.out
(dbx) stopi at 0x11230  Set breakpoint at address
(2) stopi at &MAIN+0x68
(dbx) run  Run program
Running: a.out
(process id 18803)
stopped in MAIN at 0x11230
MAIN+0x68:  fdivs  %f3, %f2, %f2
(dbx) where  Shows the line number of the exception
 =>[1] MAIN(), line 7 in "LocExcHan.F"
(dbx) list 7  Displays the source code line
    7      t = r/s
(dbx) cont  Continue after breakpoint, enter handler routine
Signal 8 code 3 at hex address 11230
abort: called
signal ABRT (Abort) in _kill at 0xef6e18a4
_kill+0x8:  bgeu _kill+0x30
Current function is exhandler
    24      CALL abort()
```

Of course, there are easier ways to determine the source line that caused the error. However, this example does serve to show the basics of exception handling.

### Disabling All Signal Handlers

With `f77`, some system signal handlers for trapping interrupts, bus errors, segmentation violations, or illegal instructions are automatically enabled by default.

Although generally you would not want to turn off this default behavior, you can do so by compiling a C program that sets the global C variable `f77_no_handlers` to 1 and linking into your executable program:

```
demo% cat NoHandlers.c
#include <stdio.h>
int  f77_no_handlers=1 ;
demo% cc -c NoHandlers.c
```

Otherwise, by default, `f77_no_handlers` is 0. The setting takes effect just before execution is transferred to the user program.
This variable is in the global name space of the program; do not use f77_no_handlers as the name of a variable anywhere else in the program.

With F95, no signal handlers are on by default.

Retrospective Summary

The ieee_retrospective function queries the floating-point status registers to find out which exceptions have accrued and a message is printed to standard error to inform you which exceptions were raised but not cleared. This function is automatically called by Fortran 77 programs at normal program termination (CALL EXIT). The message typically looks like this; the format may vary with each compiler release:

Note: IEEE floating-point exception flags raised:
  Division by Zero;
IEEE floating-point exception traps enabled:
  inexact; underflow; overflow; invalid operation;
See the Numerical Computation Guide, ieee_flags(3M),
  ieee_handler(3M)

Fortran 95 programs do not call ieee_retrospective automatically. A Fortran 95 program would need to call ieee_retrospective explicitly (and link with -lf77compat).

Debugging IEEE Exceptions

In most cases, the only indication that any floating-point exceptions (such as overflow, underflow, or invalid operation) have occurred is the retrospective summary message at program termination. Locating where the exception occurred requires exception trapping be enabled. This can be done by either compiling with the -ftrap=common option or by establishing an exception handler routine with ieee_handler(). With exception trapping enabled, run the program from dbx or the Sun WorkShop, using the dbx catch FPE command to see where the error occurs.

The advantage of recompiling with -ftrap=common is that the source code need not be modified to trap the exceptions. However, by calling ieee_handler() you can be more selective as to which exceptions to look at.
Example: Recompiling with \texttt{-ftrap=common} and using \texttt{dbx}:

\begin{verbatim}
demo\% f77 -g -ftrap=common -silent myprogram.f
demo\% dbx a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for libF77.so.3
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19739)
signal FPE (floating point divide by zero) in MAIN at line 212 in
file "myprogram.f"
212       Z = X/Y
(dbx) print Y
y = 0.0
(dbx)
\end{verbatim}

If you find that the program terminates with overflow and other exceptions, you can locate the first overflow specifically by calling \texttt{ieee_handler()} to trap just overflows. This requires modifying the source code of at least the main program, as shown in the following example.
Example: Locate an overflow when other exceptions occur:

```plaintext
demo% cat myprog.F
#include "f77_floatingpoint.h"
program myprogram
...
   ier = ieee_handler(‘set’,’overflow’,SIGFPE_ABORT)
...
demo% f77 -g -silent myprog.F
demo% dbx a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for libF77.so.3
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19793)
signal FPE (floating point overflow) in MAIN at line 55 in file "myprog.F"
   55      w = rmax * 200.      ! Cause of the overflow
(dbx) cont          ! Continue execution to completion
Note: IEEE floating-point exception flags raised:
   Inexact; Division by Zero; Underflow; ! There were other exceptions
IEEE floating-point exception traps enabled:
   overflow;
See the Numerical Computation Guide...
execution completed, exit code is 0
(dbx)
```

To be selective, the example introduces the `#include`, which required renaming the source file with a .F suffix and calling `ieee_handler()`. You could go further and create your own handler function to be invoked on the overflow exception to do some application-specific analysis and print intermediary or debug results before aborting.
Further Numerical Adventures

This section addresses some real world problems that involve arithmetic operations that may unwittingly generate invalid, division by zero, overflow, underflow, or inexact exceptions.

For instance, prior to the IEEE standard, if you multiplied two very small numbers on a computer, you could get zero. Most mainframes and minicomputers behaved that way. With IEEE arithmetic, gradual underflow expands the dynamic range of computations.

For example, consider a 32-bit processor with 1.0E-38 as the machine’s epsilon, the smallest representable value on the machine. Multiply two small numbers:

\[
\begin{align*}
a &= 1.0E-30 \\
b &= 1.0E-15 \\
x &= a \times b
\end{align*}
\]

In older arithmetic, you would get 0.0, but with IEEE arithmetic and the same word length, you get 1.40130E-45. Underflow tells you that you have an answer smaller than the machine naturally represents. This result is accomplished by “stealing” some bits from the mantissa and shifting them over to the exponent. The result, a denormalized number, is less precise in some sense, but more precise in another. The deep implications are beyond this discussion. If you are interested, consult Computer, January 1980, Volume 13, Number 1, particularly J. Coonen’s article, “Underflow and the Denormalized Numbers.”

Most scientific programs have sections of code that are sensitive to roundoff, often in an equation solution or matrix factorization. Without gradual underflow, programmers are left to implement their own methods of detecting the approach of an inaccuracy threshold. Otherwise they must abandon the quest for a robust, stable implementation of their algorithm.

For more details on these topics, see the Sun WorkShop Numerical Computation Guide.

Avoiding Simple Underflow

Some applications actually do a lot of computation very near zero. This is common in algorithms computing residuals or differential corrections. For maximum numerically safe performance, perform the key computations in extended precision arithmetic. If the application is a single-precision application, you can perform key computations in double precision.
Example: A simple dot product computation in single precision:

```fortran
sum = 0
DO i = 1, n
    sum = sum + a(i) * b(i)
END DO
```

If \( a(i) \) and \( b(i) \) are very small, many underflows occur. By forcing the computation to double precision, you compute the dot product with greater accuracy and do not suffer underflows:

```fortran
DOUBLE PRECISION sum
DO i = 1, n
    sum = sum + dble(a(i)) * dble(b(i))
END DO
result = sum
```

*On SPARC platforms:* You can force a SPARC processor to behave like an older system with respect to underflow (Store Zero) by adding a call to the library routine `nonstandard_arithmetic()` or by compiling the application’s main program with the `-fns` option.

**Continuing With the Wrong Answer**

You might wonder why you would continue a computation if the answer is clearly wrong. IEEE arithmetic allows you to make distinctions about what kind of wrong answers can be ignored, such as NaN or Inf. Then decisions can be made based on such distinctions.

For an example, consider a circuit simulation. The only variable of interest (for the sake of argument) from a particular 50-line computation is the voltage. Further, assume that the only values that are possible are +5v, 0, -5v.

It is possible to carefully arrange each part of the calculation to coerce each sub-result to the correct range:

```fortran
if computed value is greater than 4.0, return 5.0
if computed value is between -4.0 and +4.0, return 0
if computed value is less than -4.0, return -5.0
```

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. The computation can be written in the obvious fashion, and only the final result need be coerced to the correct value—since $\text{Inf}$ 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 in executing, since you don’t do unneeded comparisons.

**SPARC: Excessive Underflow**

If two very small numbers are multiplied, the result underflows.

If you know in advance that the operands in a multiplication (or subtraction) may be small and underflow is likely, run the calculation in double precision and convert the result to single precision later.

For example, a dot product loop like this:

```fortran
real sum, a(maxn), b(maxn)
...
do i =1, n
   sum = sum + a(i)*b(i)
endo
```

where the $a(*)$ and $b(*)$ are known to have small elements, should be run in double precision to preserve numeric accuracy:

```fortran
real a(maxn), b(maxn)
double sum
...
do i =1, n
   sum = sum + a(i)*dble(b(i))
endo
```

Doing so may also improve performance due to the software resolution of excessive underflows caused by the original loop. However, there is no hard and fast rule here; experiment with your intensely computational code to determine the most profitable solutions.
Interval Arithmetic

The Sun WorkShop 6 Fortran 95 compiler f95 supports intervals as an intrinsic data type. An interval is the closed compact set: 

\[ [a, b] = \{ z \mid a \leq z \leq b \} \]

defined by a pair of numbers, \( a \leq b \). Intervals can be used to:

- Solve nonlinear problems
- Perform rigorous error analysis
- Detect sources of numerical instability

By introducing intervals as an intrinsic data type to Fortran 95, all of the applicable syntax and semantics of Fortran 95 become immediately available to the developer. Besides the INTERVAL data types, f95 includes the following interval extensions to Fortran 95:

- Three classes of INTERVAL relational operators:
  - Certainly
  - Possibly
  - Set

- Intrinsic INTERVAL-specific operators, such as INF, SUP, WID, and HULL
- INTERVAL input/output edit descriptors, including single-number input/output
- Interval extensions to arithmetic, trigonometric, and other mathematical functions
- Expression context-dependent INTERVAL constants
- Mixed-mode interval expression processing

To use the Fortran 95 interval-specific features, specify -xia or -xinterval in the f95 command line.

For detailed information on interval arithmetic in Fortran 95, see the Interval Arithmetic Programming Reference.
Porting

This chapter discusses the porting of programs from other dialects of Fortran to Sun compilers. VAX VMS Fortran programs compile almost exactly as is with Sun f77; this is discussed further in the chapter on VMS extensions in the FORTRAN 77 Language Reference Manual.

Note – Porting issues bear mostly upon Fortran 77 programs. The Sun WorkShop Fortran 95 compiler, f95, incorporates few nonstandard extensions, and these are described in the Fortran User’s Guide.

Time and Date Functions

Library functions that return the time of day or elapsed CPU time vary from system to system.

The following time functions are not supported directly in the Sun Fortran libraries, but you can write subroutines to duplicate their functions:

- Time-of-day in 10h format
- Date in A10 format
- Milliseconds of job CPU time
- Julian date in ASCII
The time functions supported in the Sun Fortran library are listed in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Man Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>Returns the number of seconds elapsed since January, 1, 1970</td>
<td>time(3F)</td>
</tr>
<tr>
<td>date</td>
<td>Returns date as a character string</td>
<td>date(3F)</td>
</tr>
<tr>
<td>fdate</td>
<td>Returns the current time and date as a character string</td>
<td>fdate(3F)</td>
</tr>
<tr>
<td>idate</td>
<td>Returns the current month, day, and year in an integer array</td>
<td>idate(3F)</td>
</tr>
<tr>
<td>itime</td>
<td>Returns the current hour, minute, and second in an integer array</td>
<td>itime(3F)</td>
</tr>
<tr>
<td>ctime</td>
<td>Converts the time returned by the time function to a character string</td>
<td>ctime(3F)</td>
</tr>
<tr>
<td>ltime</td>
<td>Converts the time returned by the time function to the local time</td>
<td>ltime(3F)</td>
</tr>
<tr>
<td>gmttime</td>
<td>Converts the time returned by the time function to Greenwich time</td>
<td>gmttime(3F)</td>
</tr>
<tr>
<td>etime</td>
<td><em>Single processor:</em> Returns elapsed user and system time for program execution. <em>Multiple processors:</em> Returns the wall clock time.</td>
<td>etime(3F)</td>
</tr>
<tr>
<td>dtime</td>
<td>Returns the elapsed user and system time since last call to dtime</td>
<td>dtime(3F)</td>
</tr>
<tr>
<td>date_and_time</td>
<td>Returns date and time in character and numeric form</td>
<td>date_and_time(3F)</td>
</tr>
</tbody>
</table>

For details, see Fortran Library Reference Manual or the individual man pages for these functions.

The routines listed in the following table provide compatibility with VMS Fortran system routines idate and time. To use these routines, you must include the -lV77 option on the $f77$ command line, in which case you also get these VMS versions instead of the standard $f77$ versions.
Table 7-2  Summary: Nonstandard VMS Fortran System Routines

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Calling Sequence</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idate</td>
<td>Date as day, month, year</td>
<td>call idate( d, m, y )</td>
<td>integer</td>
</tr>
<tr>
<td>time</td>
<td>Current time as hhmmss</td>
<td>call time( t )</td>
<td>character*8</td>
</tr>
</tbody>
</table>

**Note** – The date(3F) routine and the VMS version of idate(3F) cannot be Year 2000 safe because they return 2-digit values for the year. Programs that compute time duration by subtracting dates returned by these routines will compute erroneous results after December 31, 1999. The Fortran 95 routine date_and_time(3F) is available for both FORTRAN 77 and Fortran 95 programs, and should be used instead. See the Fortran Library Reference Manual for details.

The error condition subroutine errsns is not provided, because it is totally specific to the VMS operating system.
Here is a simple example of the use of these time functions (TestTim.f):

```fortran
subroutine startclock
    common / myclock / mytime
    integer mytime, time
    mytime = time()
    return
end

function wallclock
    integer wallclock
    common / myclock / mytime
    integer mytime, time, newtime
    newtime = time()
    wallclock = newtime - mytime
    mytime = newtime
    return
end

integer wallclock, elapsed
character*24 greeting
real dtime, timediff, timearray(2)
c      print a heading
    call fdate( greeting )
    print*, "Hello, Time Now Is: ", greeting
    print*, "See how long 'sleep 4' takes, in seconds"
    call startclock
    call system( 'sleep 4' )
    elapsed = wallclock()
    print*, "Elapsed time for sleep 4 was: ", elapsed," seconds"
c      now test the cpu time for some trivial computing
    timediff = dtime( timearray )
    q = 0.01
    do 30 i = 1, 1000
       q = atan( q )
    30 continue
    timediff = dtime( timearray )
    print*, "atan(q) 1000 times took: ", timediff," seconds"
end
```

Running this program produces the following results:

demo% TimeTest
Hello, Time Now Is: Mon Feb 12 11:53:54 1996
See how long 'sleep 4' takes, in seconds
Elapsed time for sleep 4 was:  5 seconds
atan(q) 1000 times took:     2.26550E-03 seconds
demo%
Formats

Some $\texttt{f77}$ and $\texttt{f95}$ format edit descriptors can behave differently on other systems. Here are some format specifiers that $\texttt{f77}$ treats differently than some other implementations:

- **A** – Alphanumeric conversion. Used with character type data elements. In 
  FORTRAN 77, this specifier worked with any variable type. $\texttt{f77}$ supports the 
  older usage, up to four characters to a word.
- **S** – Suppresses newline character output.
- **R** – Sets an arbitrary radix for the $I$ formats that follow in the descriptor.
- **SU** – Selects unsigned output for following $I$ formats. For example, you can 
  convert output to either hexadecimal or octal with the following formats, instead 
  of using the $Z$ or $O$ edit descriptors:

  ```fortran
  10    FORMAT( SU, 16R, I4 )
  20    FORMAT( SU, 8R, I4 )
  ```

Carriage-Control

Fortran carriage-control grew out of the capabilities of the equipment used when 
Fortran was originally developed. For similar historical reasons, an operating system 
derived from the UNIX operating system, does not have Fortran carriage control, but 
you can simulate it in two ways.

- Use the $\texttt{asa}$ filter to transform Fortran carriage-control conventions into the 
  UNIX carriage-control format (see the $\texttt{asa}$ (1) man page) before printing files 
  with the $\texttt{lpr}$ command.
- **$\texttt{f77}$**: For simple jobs, use 
  \[
  \texttt{OPEN(N, FORM='PRINT')} \]
  to enable single or double 
  spacing, formfeed, and stripping off of column one. It is legal to reopen unit 6 
  to change the form parameter to PRINT. For example:

  ```fortran
  OPEN( 6, FORM='PRINT')
  ```

You can use $\texttt{lp}(1)$ to print a file that is opened in this manner.
Working With Files

Early Fortran systems did not use named files, but did provide a command line mechanism to equate actual file names with internal unit numbers. This facility can be emulated in a number of ways, including standard UNIX redirection.

Example: Redirecting stdin to redir.data (using csh(I)):

```
demo% cat redir.data
  9  9.9

demo% cat redir.f
  read(*,*) i, z
  print *, i, z
  stop
  end

demo% f77 -silent -o redir redir.f

demo% redir < redir.data
```

Porting From Scientific Mainframes

If the application code was originally developed for 64-bit (or 60-bit) mainframes such as CRAY or CDC, you might want to compile these codes with the following options when porting to an UltraSPARC-II platform, for example:

```
-fast -xarch=v9a -xchip=ultra2 \
-xtypemap=real:64,double:64,integer:64
```

These options automatically promote all default REAL variables and constants to REAL*8, and COMPLEX to COMPLEX*16. Only undeclared variables or variables declared as simply REAL or COMPLEX are promoted; variables declared explicitly (for example, REAL*4) are not promoted. All single-precision REAL constants are also promoted to REAL*8. (Set -xarch and -xchip appropriately for the target platform.) To also promote default DOUBLE PRECISION data to REAL*16, change the double:64 to double:128 in the -xtypemap example.

The -xtypemap option, is preferred over -dbl and -r8 and -i2. See the Fortran User’s Guide and the f77(1) or f95(1) man pages for details.
To further recreate the original mainframe environment, it is probably preferable to stop on overflows, division by zero, and invalid operations. Compile the main program with `-ftrap=common` to ensure this.

## Data Representation

The *FORTRAN 77 Language Reference Manual*, *Fortran User’s Guide*, and the Sun *Numerical Computation Guide* discuss in detail the hardware representation of data objects in Fortran. Differences between data representations across systems and hardware platforms usually generate the most significant portability problems.

The following issues should be noted:

- Sun adheres to the IEEE Standard 754 for floating-point arithmetic. Therefore, the first four bytes in a `REAL*8` are not the same as in a `REAL*4`.
- The default sizes for reals, integers, and logicals are described in the FORTRAN 77 standard, except when these default sizes are changed by the `-xtypemap=` option (or by `-i2`, `-dbl`, or `-r8`).
- Character variables can be freely mixed and equivalenced to variables of other types, but be careful of potential alignment problems.
- `f77` IEEE floating-point arithmetic does raise exceptions on overflow or divide by zero but does not signal `SIGFPE` or trap by default. It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. This is explained in the *Floating Point Arithmetic* chapter of this Guide.
- The extreme finite, normalized values can be determined. See `libm_single(3F)` and `libm_double(3F)`. The indeterminate forms can be written and read, using formatted and list-directed I/O statements.

## Hollerith Data

Many “dusty-deck” Fortran applications store Hollerith ASCII data into numerical data objects. With the 1977 Fortran standard (and Fortran 95), the `CHARACTER` data type was provided for this purpose and its use is recommended. You can still initialize variables with the older Fortran Hollerith (nH) feature, but this is not standard practice. The following table indicates the maximum number of characters
that will fit into certain data types. (In this table, boldfaced data types indicate default types subject to promotion by the \texttt{f77} command-line flags \texttt{-dbl}, \texttt{-r8}, or \texttt{-xtypemap=}).

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Maximum Number of Standard ASCII Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No \texttt{-i2}, \texttt{-i4}, \texttt{-r8}, \texttt{-dbl}</td>
</tr>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>16</td>
</tr>
<tr>
<td>COMPLEX*32</td>
<td>32</td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>16</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td>8</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*1</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*8</td>
<td>8</td>
</tr>
<tr>
<td>REAL</td>
<td>4</td>
</tr>
<tr>
<td>REAL*4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*8</td>
<td>8</td>
</tr>
<tr>
<td>REAL*16</td>
<td>16</td>
</tr>
</tbody>
</table>

When storing standard ASCII characters with normal Fortran:
- With \texttt{-r8}, unspecified size INTEGER and LOGICAL do not hold double.
- With \texttt{-dbl}, unspecified size INTEGER and LOGICAL do hold double.

The storage is allocated with both options, but it is unavailable in normal Fortran with \texttt{-r8}.

Options \texttt{-i2}, \texttt{-r8}, and \texttt{-dbl} are now considered obsolete; use \texttt{-xtypemap} instead.
Example: Initialize variables with Hollerith:

```fortran
 demo% cat FourA8.f
  double complex x(2)
  data x /16Habcdefgijklmnop, 16Hqrstuvwxyz012345/
  write( 6, '(4A8, "!")' ) x
 end

demo% f77 -silent -o FourA8 FourA8.f

demo% FourA8
abcdefgijklmnopqrstuvwxyz012345!
demo%
```

If you pass Hollerith constants as arguments, or if you use them in expressions or comparisons, they are interpreted as character-type expressions.

If needed, you can initialize a data item of a compatible type with a Hollerith and then pass it to other routines.
Example:

```fortran
program respond
  integer yes, no
  integer ask
  data yes, no / 3hyes, 2hno /

  if ( ask() .eq. yes ) then
    print *, 'You may proceed!
  else
    print *, 'Request Rejected!
  endif
end

integer function ask()
  double precision solaris, response
  integer yes, no
  data yes, no / 3hyes, 2hno /
  data solaris/ 7hSOLARIS/
  10      format( "What system? ", $ )
  20      format( a8 )

  write( 6, 10 )
  read ( 5, 20 ) response
  ask = no
  if ( response .eq. solaris ) ask = yes
return
end
```

Nonstandard Coding Practices

As a general rule, porting an application program from one system and compiler to another can be made easier by eliminating any nonstandard coding. Optimizations or work-arounds that were successful on one system might only obscure and confuse compilers on other systems. In particular, optimized hand-tuning for one particular architecture can cause degradations in performance elsewhere. This is discussed later in the chapters on performance and tuning. However, the following issues are worth considering with regards to porting in general.
Uninitialized Variables

Some systems automatically initialize local and COMMON variables to zero or some "not-a-number" (NaN) value. However, there is no standard practice, and programs should not make assumptions regarding the initial value of any variable. To assure maximum portability, a program should initialize all variables.

Aliasing Across Calls

Aliasing occurs when the same storage address is referenced by more than one name. This happens when actual arguments to a subprogram overlap between themselves or between COMMON variables within the subprogram. For example, arguments X and Z refer to the same storage locations, as do B and H:

```
COMMON /INS/B(100)
REAL S(100), T(100)
...
CALL SUB(S,T,S,B,100)
...
SUBROUTINE SUB(X,Y,Z,H,N)
REAL X(N), Y(N), Z(N), H(N)
COMMON /INS/B(100)
...
```

Avoid aliasing in this manner in all portable code. The results on some systems and with higher optimization levels could be unpredictable.

Obscure Optimizations

Legacy codes may contain source-code restructurings of ordinary computational DO loops intended to cause older vectorizing compilers to generate optimal code for a particular architecture. In most cases, these restructurings are no longer needed and may degrade the portability of a program. Two common restructurings are strip-mining and loop unrolling.
Strip-Mining

Fixed-length vector registers on some architectures led programmers to manually “strip-mine” the array computations in a loop into segments:

```
REAL TX(0:63)
...
DO IOUTER = 1,NX,64
  DO IINNER = 0,63
    TX(IINNER) = AX(IOUTER+IINNER) * BX(IOUTER+IINNER)/2.
    QX(IOUTER+IINNER) = TX(IINNER)**2
  END DO
END DO
```

Strip-mining is no longer appropriate with modern compilers; the loop can be written much less obscurely as:

```
DO IX = 1,N
  TX = AX(IX)*BX(IX)/2.
  QX(IX) = TX**2
END DO
```
Loop Unrolling

Unrolling loops by hand was a typical source-code optimization technique before compilers were available that could perform this restructuring automatically. A loop written as:

\[
\begin{align*}
\text{DO} & \quad K = 1, N-5, 6 \\
\text{DO} & \quad J = 1, N \\
\text{DO} & \quad I = 1, N \\
A(I,J) & = A(I,J) + B(I,K) \cdot C(K,J) \\
& \quad + B(I,K+1) \cdot C(K+1,J) \\
& \quad + B(I,K+2) \cdot C(K+2,J) \\
& \quad + B(I,K+3) \cdot C(K+3,J) \\
& \quad + B(I,K+4) \cdot C(K+4,J) \\
& \quad + B(I,K+5) \cdot C(K+5,J) \\
\text{END DO} & \\
\text{END DO} & \\
\text{END DO} & \\
\text{DO} & \quad K = K, N \\
\text{DO} & \quad J = 1, N \\
\text{DO} & \quad I = 1, N \\
A(I,J) & = A(I,J) + B(I,KK) \cdot C(KK,J) \\
\text{END DO} & \\
\text{END DO} & \\
\text{END DO} & \\
\end{align*}
\]

should be rewritten the way it was originally intended:

\[
\begin{align*}
\text{DO} & \quad K = 1, N \\
\text{DO} & \quad J = 1, N \\
\text{DO} & \quad I = 1, N \\
A(I,J) & = A(I,J) + B(I,K) \cdot C(K,J) \\
\text{END DO} & \\
\text{END DO} & \\
\text{END DO} & \\
\end{align*}
\]
Troubleshooting

Here are a few suggestions for what to try when programs ported to Sun Fortran do not run as expected.

Results Are Close, but Not Close Enough

Try the following:

- Pay attention to the size and the engineering units. Numbers very close to zero can appear to be different, but the difference is not significant, especially if this number is the difference between two large numbers, such as the distance across the continent in feet, as calculated on two different computers. For example, 1.9999999e-30 is very near -9.9992112e-33, even though they differ in sign.

VAX math is not as good as IEEE math, and even different IEEE processors may differ. This is especially true if the mathematics involves many trigonometric functions. These functions are much more complicated than one might think, and the standard defines only the basic arithmetic functions. There can be subtle differences, even between IEEE machines. Review the Floating-Point Arithmetic chapter in this Guide.

- Try running with a call nonstandard_arithmetic(). Doing so can also improve performance considerably, and make your Sun workstation behave more like a VAX system. If you have access to a VAX or some other system, run it there also. It is quite common for many numerical applications to produce slightly different results on each floating-point implementation.

- Check for NaN, +Inf, and other signs of probable errors. See the Floating-Point Arithmetic chapter in this Guide, or the man page ieee_handler(3m) for instructions on how to trap the various exceptions. On most machines, these exceptions simply abort the run.

- Two numbers can differ by $6 \times 10^{29}$ and still have the same floating-point form. Here is an example of different numbers, with the same representation:

```fortran
real*4 x,y
x=99999990e+29
y=99999996e+29
write (*,10) x, x
10   format('99,999,990 x 10^29 = ', e14.8, ' = ', z8)
write(*,20) y, y
20   format('99,999,996 x 10^29 = ', e14.8, ' = ', z8)
end
```
The output is:

\[
\begin{align*}
99,999,990 \times 10^{29} &= 0.99999993 \times 10^{37} = 7CF0BDC1 \\
99,999,996 \times 10^{29} &= 0.99999993 \times 10^{37} = 7CF0BDC1
\end{align*}
\]

In this example, the difference is \(6 \times 10^{29}\). The reason for this indistinguishable, wide gap is that in IEEE single-precision arithmetic, you are guaranteed only six decimal digits for any one decimal-to-binary conversion. You may be able to convert seven or eight digits correctly, but it depends on the number.

### Program Fails Without Warning

If the program fails without warning and runs different lengths of time between failures, then:

- Compile with minimal optimization (\(-O1\)). If the program then works, compile only selective routines with higher optimization levels.
- Understand that optimizers must make assumptions about the program. Nonstandard coding or constructs can cause problems. Almost no optimizer handles all programs at all levels of optimization.
Performance Profiling

This chapter describes how to measure and display program performance. Knowing where a program is spending most of its compute cycles and how efficiently it uses system resources is a prerequisite for performance tuning.

Sun WorkShop Performance Analyzer

Sun WorkShop Performance Analyzer provides a sophisticated pair of tools for collecting and analyzing program performance data:

- The Sampling Collector collects performance data (statistical profiles of call stacks, thread-synchronization delay events, hardware-counter overflow profiles, address space data, and summary information for the operating system), and stores it in an experiment file.
- The Sampling Analyzer displays the data recorded by the Sampling Collector, so you can examine the information. The analyzer processes the data and displays various metrics of performance, at function, caller-callee, source-line, disassembly-instruction, and program levels.

The Sampling Analyzer can also help you to fine-tune your application’s performance, by creating a mapfile you can use to improve the order of function loading in the application address space.

The Collector and Analyzer are designed for use by any software developer, even if performance tuning is not the developer’s main responsibility.

Command-line equivalents of the Collector and Analyzer are available:

- dbx includes a data-collection feature that has the same functionality as the Collector.
- The command-line utility er_print(1), which prints out an ASCII version of the various Analyzer displays, operates as a command-line sampling analyzer.
Details can be found in the Sun WorkShop manual *Analyzing Program Performance With Sun WorkShop.*

### The `time` Command

The simplest way to gather basic data about program performance and resource utilization is to use the `time` command or, in `csh`, the `set time` command.

Running the program with the `time` command prints a line of timing information on program termination.

```bash
demo% time myprog
   The Answer is: 543.01
  6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
demo%
```

The interpretation is:

- `user` – 6.5 seconds in user code, approximately
- `system` – 17.1 seconds in system code for this task, approximately
- `wallclock` – 1 minute 16 seconds to complete
- `resources` – 31% of system resources dedicated to this program
- `memory` – 11 Kilobytes of shared program memory, 21 kilobytes of private data
- `I/O` – 354 reads, 210 writes
- `paging` – 135 page faults, 0 swapouts

### Multiprocessor Interpretation of `time` Output

Timing results are interpreted in a different way when the program is run in parallel in a multiprocessor environment. Since `/bin/time` accumulates the user time on different threads, only wall clock time is used.

Since the user time displayed includes the time spent on all the processors, it can be quite large and is not a good measure of performance. A better measure is the real time, which is the wall clock time. This also means that to get an accurate timing of a parallelized program you must run it on a quiet system dedicated to just your program.
The *gprof* Profiling Command

The *gprof*(1) command provides a detailed postmortem analysis of program timing at the subprogram level, including how many times a subprogram was called, who called it, whom it called, and how much time was spent in the routine and by the routines it called.

To enable *gprof* profiling, compile and link the program with the `-pg` option:

```
demo% f77 -o Myprog -fast -pg Myprog.f ...
demo% Myprog
demo% gprof Myprog
```

The program must complete normally for *gprof* to obtain meaningful timing information.

At program termination, the file `gmon.out` is automatically written in the working directory. This file contains the profiling data that will be interpreted by *gprof*.

Invoking *gprof* produces a report on standard output. An example is shown on the next pages. Not only the routines in your program are listed but also the library procedures and the routines they call.

The report is mostly two profiles of how the total time is distributed across the program procedures: the call graph and the flat profile. They are preceded by an explanation of the column labels, followed by an index. (The *gprof* `-b` option eliminates the explanatory text; see the *gprof*(1) man page for other options that can be used to limit the amount of output generated.)

In the graph profile, each procedure (subprogram, procedure) is presented in a call-tree representation. The line representing a procedure in its call-tree is called the *function line*, and is identified by an index number in the leftmost column, within square brackets; the lines above it are the *parent lines*; the lines below it, the *descendant lines*. 
The call graph profile is followed by a flat profile that provides a routine-by-routine overview. An (edited) example of gprof output follows.

**Note** – User-defined functions appear with their Fortran names followed by an underscore. Library routines appear with leading underscores.
The call graph profile:

```plaintext
granularity: each sample hit covers 2 byte(s) for 0.08% of 12.78 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self descendents</th>
<th>called/total called+self called/total name</th>
<th>parents index</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>12.66</td>
<td>1/1</td>
<td>main [1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.92</td>
<td>10.99</td>
<td>1000/1000</td>
<td>differ_ [4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.62</td>
<td>0.00</td>
<td>2000/2001</td>
<td>code_ [9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.11</td>
<td>0.00</td>
<td>1000/1000</td>
<td>shock_ [11]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>0.00</td>
<td>1000/1000</td>
<td>bndry_ [14]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td>init_ [24]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>2/2</td>
<td>output_ [40]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td>input_ [47]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-----------------------------------------------

| 0.92 | 10.99 | 1000/1000 | MAIN_ [3] |
| 0.92 | 10.99 | 1000      | differ_ [4] |
| 1.11 | 4.52 | 3000/3000 | deriv_ [7] |
| 1.29 | 2.91 | 3000/6000 | cheb1_ [5] |
| 1.17 | 0.00 | 3000/3000 | dissip_ [8] |

-----------------------------------------------

| 1.29 | 2.91 | 3000/6000 | deriv_ [7] |
| 1.29 | 2.91 | 3000/6000 | differ_ [4] |
| 2.58 | 5.81 | 6000      | cheb1_ [5] |
| 5.81 | 0.00 | 6000/6000 | fftb_ [6] |
| 0.00 | 0.00 | 128/321   | cos [21]   |
| 0.00 | 0.00 | 128/192   | __sin [279] |

-----------------------------------------------

| 5.81 | 0.00 | 6000/6000 | cheb1_ [5] |
| 5.81 | 0.00 | 6000      | fftb_ [6] |
| 0.00 | 0.00 | 64/321    | cos [21]   |
| 0.00 | 0.00 | 64/192    | __sin [279] |

-----------------------------------------------

...
The flat profile overview:

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>45.2</td>
<td>5.81</td>
<td>5.81</td>
<td>6000</td>
</tr>
<tr>
<td>20.1</td>
<td>8.39</td>
<td>2.5</td>
<td>6000</td>
</tr>
<tr>
<td>9.1</td>
<td>9.56</td>
<td>1.17</td>
<td>3000</td>
</tr>
<tr>
<td>8.6</td>
<td>10.67</td>
<td>1.11</td>
<td>3000</td>
</tr>
<tr>
<td>7.1</td>
<td>11.58</td>
<td>0.92</td>
<td>1000</td>
</tr>
<tr>
<td>4.8</td>
<td>12.20</td>
<td>0.62</td>
<td>2001</td>
</tr>
<tr>
<td>2.5</td>
<td>12.53</td>
<td>0.33</td>
<td>69000</td>
</tr>
<tr>
<td>0.9</td>
<td>12.64</td>
<td>0.11</td>
<td>1000</td>
</tr>
</tbody>
</table>

- **Function Line.**

The function line [5] in the preceding example reveals that:

- `cheb1` was called 6000 times—3000 from `deriv`, 3000 from `diffr`.
- 2.58 seconds were spent in `cheb1` itself.
- 5.81 seconds were spent in routines called by `cheb1`.
- 65.7% of the execution time of the program was within `cheb1`.

- **Parent Lines.**

The parent lines above [5] indicate that `cheb1` was called from two routines, `deriv` and `diffr`. The timings on these lines show how much time was spent in `cheb1` when it was called from each of these routines.

- **Descendant Lines.**

The lines below the function line indicate the routines called from `cheb1`, `fftb`, `sin`, and `cos`. The library sine function is called indirectly.

- **Flat Profile.**

Function names appear on the right. The profile is sorted by percentage of total execution time.
Overhead and Other Considerations

Profiling (compiling with the \texttt{-pg} option) may greatly increase the running time of a program. This is due to the extra overhead required to clock program performance and subprogram calls. Profiling tools like \texttt{gprof} attempt to subtract an approximate overhead factor when computing relative runtime percentages. All other timings shown may not be accurate due to UNIX and hardware timekeeping inaccuracies.

Programs with short execution times are the most difficult to profile because the overhead may be a significant fraction of the total execution time. The best practice is to choose input data for the profiling run that will result in a realistic test of the program's performance. If this is not possible, consider enclosing the main computational part of the program within a loop that effectively runs the program \( N \) times. Estimate actual performance by dividing the profile results by \( N \).

The Fortran library includes two routines that return the total time used by the calling process. See the man pages for \texttt{dtime(3F)} and \texttt{etime(3F)}.

Also, \texttt{gprof} can give misleading results. A well-known limitation is that \texttt{gprof} cannot differentiate time spent in a function called from more than one caller. For example, it may be that function \texttt{FU} takes much more time when called from routine \texttt{BAR} than from any other routine, and knowing this could suggest to you a significant restructuring of the program and better performance. Unfortunately, the results shown by \texttt{gprof} average the total time spent in \texttt{FU} over all calls, obscuring this valuable bit of information. The Sun WorkShop Performance Analyzer provides much more detailed and useful information if you intend to do serious performance analysis of a program, and should be used instead.

The \texttt{tcov} Profiling Command

The \texttt{tcov(1)} command, when used with programs compiled with the \texttt{-a}, \texttt{-xa}, or \texttt{-xprofile=tcov} options, produces a statement-by-statement profile of the source code showing which statements executed and how often. It also gives a summary of information about the basic block structure of the program.

There are two implementations of \texttt{tcov} coverage analysis. The original \texttt{tcov} is invoked by the \texttt{-a} or \texttt{-xa} compiler options. Enhanced statement level coverage is invoked by the \texttt{-xprofile=tcov} compiler option and the \texttt{-x tcov} option. In either case, the output is a copy of the source files annotated with statement execution counts in the margin. Although these two versions of \texttt{tcov} are essentially the same as far as the Fortran user is concerned (most of the enhancements apply to C++ programs), there will be some performance improvement with the newer style.
Note – The code coverage report produced by tcov will be unreliable if the
compiler has inlined calls to routines. The compiler inlines calls whenever
appropriate at optimization levels above -O3, and according to the -inline option.
With inlining, the compiler replaces a call to a routine with the actual code for the
called routine. And, since there is no call, references to those inlined routines will not
be reported by tcov. Therefore, to get an accurate coverage report, do not enable
compiler inlining.

“Old Style” t cov Coverage Analysis

Compile the program with the -a (or -xa) option. This produces the file $TCOVDIR/
file.d for each source .f file in the compilation. (If environment variable $TCOVDIR
is not set at compile time, the .d files are stored in the current directory.)

Run the program (execution must complete normally). This produces updated
information in the .d files. To view the coverage analysis merged with the
individual source files, run t cov on the source files. The annotated source files are
named $TCOVDIR/file.tcov for each source file.

The output produced by t cov shows the number of times each statement was
actually executed. Statements that were not executed are marked with ####-> to the
left of the statement.
Here is a simple example:

demo% f77 -a -o onetwo -silent one.f two.f
demo% onetwo
... output from program
demo% t cov one.f two.f
demo% cat one.t cov two.t cov

```
program one
  1 -> do i=1,10
  10 -> call two(i)
   end do
  1 -> end
```

Top 10 Blocks
<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

3      Basic blocks in this file
3      Basic blocks executed
100.00 Percent of the file executed
12     Total basic block executions
4.00   Average executions per basic block

subroutine two(i)
  10 -> print*, "two called", i
   return
   end

Top 10 Blocks
<table>
<thead>
<tr>
<th>Line</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

1      Basic blocks in this file
1      Basic blocks executed
100.00 Percent of the file executed
10     Total basic block executions
10.00  Average executions per basic block

demo%
"New Style" Enhanced `tcov` Analysis

To use new style `tcov`, compile with `-xprofile=tcov`. When the program is run, coverage data is stored in `program.profile/tcovd`, where `program` is the name of the executable file. (If the executable were `a.out`, `a.out.profile/tcovd` would be created.)

Run `tcov -x dirname source_files` to create the coverage analysis merged with each source file. The report is written to `file.tcov` in the current directory.

Running a simple example:

```bash
demo% f77 -o onetwo -silent -xprofile=tcov one.f two.f
demo% onetwo
... output from program
demo% tcov -x onetwo.profile one.f two.f
demo% cat one.f.tcov two.f.tcov
```

```
program one
  1 ->    do i=1,10
  10 ->    call two(i)
            end do
  1 ->    end
  .....etc
```

Environment variables `$SUN_PROFDATA` and `$SUN_PROFDATA_DIR` can be used to specify where the intermediary data collection files are kept. These are the `*.d` and `tcovd` files created by old and new style `tcov`, respectively.

Each subsequent run accumulates more coverage data into the `tcovd` file. Data for each object file is zeroed out the first time the program is executed after the corresponding source file has been recompiled. Data for the entire program is zeroed out by removing the `tcovd` file.

These environment variables can be used to separate the collected data from different runs. With these variables set, the running program writes execution data to the files in `$SUN_PROFDATA_DIR/$SUN_PROFDATA/`.

Similarly, the directory that `tcov` reads is specified by `tcov -x $SUN_PROFDATA`. If `$SUN_PROFDATA_DIR` is set, `tcov` will prepend it, looking for files in `$SUN_PROFDATA_DIR/$SUN_PROFDATA/`, and not in the working directory.

For the details, see the `tcov(1)` man page.
f77 I/O Profiling

You can obtain a report about how much data was transferred by your program. For each Fortran unit, the report shows the file name, the number of I/O statements, the number of bytes, and some statistics on these items.

To obtain an I/O profiling report, insert calls to the library routines start_iostats and end_iostats around the parts of the program you wish to measure. (A call to end_iostats is required if the program terminates with an END or STOP statement rather than a CALL EXIT.)

Note – The I/O statements profiled are: READ, WRITE, PRINT, OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, and REWIND. The runtime system opens stdin, stdout, and stderr before the first executable statement of your program, so you must explicitly reopen these units after the call to start_iostats.

Example: Profile stdin, stdout, and stderr:

```fortran
EXTERNAL start_iostats
.
  CALL start_iostats
  OPEN(5)
  OPEN(6)
  OPEN(0)
```

If you want to measure only part of the program, call end_iostats to stop the process. A call to end_iostats may also be required if your program terminates with an END or STOP statement rather than CALL EXIT.

The program must be compiled with the -pg option. When the program terminates, the I/O profile report is produced on the file name.io_stats, where name is the name of the executable file.
Here is an example:

**demo** % *f77 -o myprog -pg -silent myprog.f*
**demo** % *myprog*

... output from program

**demo** % *cat myprog.io_stats*

INPUT REPORT

<table>
<thead>
<tr>
<th>1. unit</th>
<th>2. file name</th>
<th>3. input data</th>
<th>4. map</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>total</td>
<td>avg</td>
<td>std dev (cnt)</td>
</tr>
<tr>
<td>0</td>
<td>stderr</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>stdin</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>stdout</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>fort.19</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>20</td>
<td>fort.20</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>21</td>
<td>fort.21</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>22</td>
<td>fort.22</td>
<td>8</td>
<td>48</td>
</tr>
</tbody>
</table>

OUTPUT REPORT

<table>
<thead>
<tr>
<th>1. unit</th>
<th>5. output data</th>
<th>6. blk size</th>
<th>7. fmt</th>
<th>8. direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>total</td>
<td>avg</td>
<td>std dev</td>
<td>(rec len)</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>40</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td>248</td>
<td>9.538</td>
<td>1.63</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>48</td>
<td>6</td>
<td>4.276</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>48</td>
<td>6</td>
<td>4.276</td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>48</td>
<td>6</td>
<td>4.276</td>
</tr>
<tr>
<td>22</td>
<td>8</td>
<td>48</td>
<td>6</td>
<td>4.276</td>
</tr>
</tbody>
</table>

...
Each pair of lines in the report displays information about an I/O unit. One section shows input operations and another shows output. The first line of a pair displays statistics on the number of data elements transferred before the unit was closed. The second row of statistics is based on the number of I/O statements processed.

In the example, there were 6 calls to write a total of 26 data elements to standard output. A total of 248 bytes was transferred. The display also shows the average and standard deviation in bytes transferred per I/O statement (9.538 and 1.63, respectively), and the average and standard deviation per I/O statement call (42.33 and 3.266, respectively).

The input report also contains a column to indicate whether a unit was memory mapped or not. If mapped, the number of mmap() calls is recorded in parentheses in the second row of the pair.

The output report indicates block sizes, formatting, and access type. A file opened for direct access shows its defined record length in parentheses in the second row of the pair.

**Note** – Compiling with environment variable `LD_LIBRARY_PATH` set might disable I/O profiling, which relies on its profiling I/O library being in a standard location.
Performance and Optimization

This chapter considers some optimization techniques that may improve the performance of numerically intense Fortran programs. Proper use of algorithms, compiler options, library routines, and coding practices can bring significant performance gains. This discussion does not discuss cache, I/O, or system environment tuning. Parallelization issues are treated in the next chapter.

Some of the issues considered here are:

- Compiler options that may improve performance
- Compiling with feedback from runtime performance profiles
- Use of optimized library routines for common procedures
- Coding strategies to improve performance of key loops

The subject of optimization and performance tuning is much too complex to be treated exhaustively here. However, this discussion should provide the reader with a useful introduction to these issues. A list of books that cover the subject much more deeply appears at the end of the chapter.

Optimization and performance tuning is an art that depends heavily on being able to determine what to optimize or tune.

Choice of Compiler Options

Choice of the proper compiler options is the first step in improving performance. Sun compilers offer a wide range of options that affect the object code. In the default case, where no options are explicitly stated on the compile command line, most options are off. To improve performance, these options must be explicitly selected.

Performance options are normally off by default because most optimizations force the compiler to make assumptions about a user’s source code. Programs that conform to standard coding practices and do not introduce hidden side effects
should optimize correctly. However, programs that take liberties with standard practices might run afoul of some of the compiler’s assumptions. The resulting code might run faster, but the computational results might not be correct.

Recommended practice is to first compile with all options off, verify that the computational results are correct and accurate, and use these initial results and performance profile as a baseline. Then, proceed in steps—recompiling with additional options and comparing execution results and performance against the baseline. If numerical results change, the program might have questionable code, which needs careful analysis to locate and reprogram.

If performance does not improve significantly, or degrades, as a result of adding optimization options, the coding might not provide the compiler with opportunities for further performance improvements. The next step would then be to analyze and restructure the program at the source code level to achieve better performance.

Performance Option Reference

The compiler options listed in the following table provide the user with a repertoire of strategies to improve the performance of a program over default compilation. Only some of the compilers’ more potent performance options appear in the table. A more complete list can be found in the Fortran User’s Guide.

<table>
<thead>
<tr>
<th>TABLE 9-1 Some Effective Performance Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>Uses a combination of optimization options together</td>
</tr>
<tr>
<td>Sets compiler optimization level to ( n )</td>
</tr>
<tr>
<td>Specifies general target hardware</td>
</tr>
<tr>
<td>Specifies a particular Instruction Set Architecture</td>
</tr>
<tr>
<td>Optimizes using performance profile data (with -O5)</td>
</tr>
<tr>
<td>Unrolls loops by ( n )</td>
</tr>
<tr>
<td>Permits simplifications and optimization of floating-point</td>
</tr>
<tr>
<td>Performs dependency analysis to optimize loops</td>
</tr>
</tbody>
</table>

Some of these options increase compilation time because they invoke a deeper analysis of the program. Some options work best when routines are collected into files along with the routines that call them (rather than splitting each routine into its own file); this allows the analysis to be global.
-fast

This single option selects a number of performance options that, working together, produce object code optimized for execution speed without an excessive increase in compilation time.

The options selected by `-fast` are subject to change from one release to another, and not all are available on each platform:

- `-native` generates code optimized for the host architecture.
- `-O5` sets optimization level.
- `-libmil` inlines calls to some simple library functions.
- `-fsimple=2` simplifies floating-point code.
- `-dalign` uses faster, double word loads and stores.
- `-xlibmopt` uses optimized `libm` math library.
- `-fns` selects non-standard floating-point mode
- `-ftrap=%none` turns off all trapping for `f77`, or
  - `-ftrap=common` selects common floating-point trapping for `f95`
- `-depend` analyzes loops for data dependencies.
- `-pad=common` improves cache performance.
- `-xvector=yes` invokes vectorized library functions in loops.

`-fast` provides a quick way to engage much of the optimizing power of the compilers. Each of the composite options may be specified individually, and each may have side effects to be aware of (discussed in the `Fortran User's Guide`).

Following `-fast` with additional options adds further optimizations. For example:

f95 -fast -xarch=v9a ...

compiles for an UltraSPARC 64-bit enabled Solaris platform.

**Note** – `-fast` includes `-dalign` and `-native`. These options may have unexpected side effects for some programs.

-0

No compiler optimizations are performed by the compilers unless a `-0` option is specified explicitly (or implicitly with macro options like `-fast`). In nearly all cases, specifying an optimization level for compilation improves program execution performance. On the other hand, higher levels of optimization increase compilation time and may significantly increase code size.

For most cases, level `-O3` is a good balance between performance gain, code size, and compilation time. Level `-O4` adds automatic inlining of calls to routines contained in the same source file as the caller routine, among other things. (See the `Fortran User’s Guide` for further information about subprogram call inlining.)
Level -O5 adds more aggressive optimization techniques that would not be applied at lower levels. In general, levels above -O3 should be specified only to those routines that make up the most compute-intensive parts of the program and thereby have a high certainty of improving performance. (There is no problem linking together parts of a program compiled with different optimization levels.)

PRAGMA OPT=n

Use the C$ PRAGMA SUN OPT=n directive to set different optimization levels for individual routines in a source file. This directive will override the -O flag on the compiler command line, but must be used with the -xmaxopt=n flag to set a maximum optimization level. See the f77(1) and f95(1) man pages for details.

Optimization With Runtime Profile Feedback

The compiler applies its optimization strategies at level O3 and above much more efficiently if combined with -xprofile=use. With this option, the optimizer is directed by a runtime execution profile produced by the program (compiled with -xprofile=collect) with typical input data. The feedback profile indicates to the compiler where optimization will have the greatest effect. This may be particularly important with -O5. Here’s a typical example of profile collection with higher optimization levels:

```
demo% f95 -o prg -fast -xprofile=collect prg.f ...
demo% prg
demo% f95 -o prgx -fast -O5 -xprofile=use:prg.profile prg.f ...
demo% prgx
```

The first compilation in the example generates an executable that produces statement coverage statistics when run. The second compilation uses this performance data to guide the optimization of the program.

(See the Fortran User’s Guide for details on -xprofile options.)

-dalign

With -dalign the compiler is able to generate double-word load/store instructions whenever possible. Programs that do much data motion may benefit significantly when compiled with this option. (It is one of the options selected by -fast.) The double-word instructions are almost twice as fast as the equivalent single word operations.
However, users should be aware that using `-dalign` (and therefore `-fast`) may cause problems with some programs that have been coded expecting a specific alignment of data in COMMON blocks. With `-dalign`, the compiler may add padding to ensure that all double (and quad) precision data (either REAL or COMPLEX) are aligned on double-word boundaries, with the result that:

- COMMON blocks might be larger than expected due to added padding.
- All program units sharing COMMON must be compiled with `-dalign` if any one of them is compiled with `-dalign`.

For example, a program that writes data by aliasing an entire COMMON block of mixed data types as a single array might not work properly with `-dalign` because the block will be larger (due to padding of double and quad precision variables) than the program expects.

`-depend`

Adding `-depend` to optimization levels `-O3` and higher (on the SPARC platform) extends the compiler’s ability to optimize DO loops and loop nests. With this option, the optimizer analyzes inter-iteration loop dependencies to determine whether or not certain transformations of the loop structure can be performed. Only loops without dependencies can be restructured. However, the added analysis might increase compilation time.

`-fsimple=2`

Unless directed to, the compiler does not attempt to simplify floating-point computations (the default is `-fsimple=0`). With the `-fast` option, `-fsimple=1` is used and some conservative assumptions are made. Adding `-fsimple=2` enables the optimizer to make further simplifications with the understanding that this might cause some programs to produce slightly different results due to rounding effects. If `-fsimple` level 1 or 2 is used, all program units should be similarly compiled to ensure consistent numerical accuracy.

`-unroll=n`

Unrolling short loops with long iteration counts can be profitable for some routines. However, unrolling can also increase program size and might even degrade performance of other loops. With `n=1`, the default, no loops are unrolled automatically by the optimizer. With `n` greater than 1, the optimizer attempts to unroll loops up to a depth of `n`. 
The compiler’s code generator makes its decision to unroll loops depending on a number of factors. The compiler might decline to unroll a loop even though this option is specified with \( n \geq 1 \).

If a DO loop with a variable loop limit can be unrolled, both an unrolled version and the original loop are compiled. A runtime test on iteration count determines whether or not executing the unrolled loop is inappropriate. Loop unrolling, especially with simple one or two statement loops, increases the amount of computation done per iteration and provides the optimizer with better opportunities to schedule registers and simplify operations. The tradeoff between number of iterations, loop complexity, and choice of unrolling depth is not easy to determine, and some experimentation might be needed.

The example that follows shows how a simple loop might be unrolled to a depth of four with \(-\text{unroll}=4\) (the source code is not changed with this option):

```
Original Loop:
  DO I=1,20000
    X(I) = X(I) + Y(I)*A(I)
  END DO

Unrolled by 4 compiles as if it were written:
  DO I=1, 19997, 4
    TEMP1 = X(I) + Y(I)*A(I)
    TEMP2 = X(I+1) + Y(I+1)*A(I+1)
    TEMP3 = X(I+2) + Y(I+2)*A(I+2)
    X(I+3) = X(I+3) + Y(I+3)*A(I+3)
    X(I) = TEMP1
    X(I+1) = TEMP2
    X(I+2) = TEMP3
  END DO
```

This example shows a simple loop with a fixed loop count. The restructuring is more complex with variable loop counts.

\(-\text{target}=\text{platform}\)

The performance of some programs might improve if the compiler has an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain could be negligible and a generic specification might be sufficient.
The *Fortran User’s Guide* lists all the system names recognized by -xtarget=. For any given system name (for example, ultra2, for UltraSPARC II™), -xtarget expands into a specific combination of -xarch, -xcache, and -xchip that properly matches that system. The optimizer uses these specifications to determine strategies to follow and instructions to generate.

The special setting -xtarget=native enables the optimizer to compile code targeted at the host system (the system doing the compilation). This is obviously useful when compilation and execution are done on the same system. When the execution system is not known, it is desirable to compile for a *generic* architecture. Therefore, -xtarget=generic is the default, even though it might produce suboptimal performance.

### Other Performance Strategies

Assuming that you have experimented with using a variety of optimization options, compiling your program and measuring actual runtime performance, the next step might be to look closely at the Fortran source program to see what further tuning can be tried.

Focusing on just those parts of the program that use most of the compute time, you might consider the following strategies:

- Replace handwritten procedures with calls to equivalent optimized libraries.
- Remove I/O, calls, and unnecessary conditional operations from key loops.
- Eliminate aliasing that might inhibit optimization.
- Rationalize tangled, spaghetti-like code to use block IF.

These are some of the good programming practices that tend to lead to better performance. It is possible to go further, hand-tuning the source code for a specific hardware configuration. However, these attempts might only further obscure the code and make it even more difficult for the compiler’s optimizer to achieve significant performance improvements. Excessive hand-tuning of the source code can hide the original intent of the procedure and could have a significantly detrimental effect on performance for different architectures.

### Using Optimized Libraries

In most situations, optimized commercial or shareware libraries perform standard computational procedures far more efficiently than you could by coding them by hand.
For example, the Sun Performance Library™ is a suite of highly optimized mathematical subroutines based on the standard LAPACK, BLAS, FFTPACK, VFFTPACK, and LINPACK libraries. Performance improvement using these routines can be significant when compared with hand coding. See the Sun Performance Library User’s Guide for details.

Eliminating Performance Inhibitors

Use the Sun WorkShop Performance Analyzer to identify the key computational parts of the program. Then, carefully analyze the loop or loop nest to eliminate coding that might either inhibit the optimizer from generating optimal code or otherwise degrade performance. Many of the nonstandard coding practices that make portability difficult might also inhibit optimization by the compiler.

Reprogramming techniques that improve performance are dealt with in more detail in some of the reference books listed at the end of the chapter. Three major approaches are worth mentioning here:

Removing I/O From Key Loops

I/O within a loop or loop nest enclosing the significant computational work of a program will seriously degrade performance. The amount of CPU time spent in the I/O library might be a major portion of the time spent in the loop. (I/O also causes process interrupts, thereby degrading program throughput.) By moving I/O out of the computation loop wherever possible, the number of calls to the I/O library can be greatly reduced.

Eliminating Subprogram Calls

Subroutines called deep within a loop nest could be called thousands of times. Even if the time spent in each routine per call is small, the total effect might be substantial. Also, subprogram calls inhibit optimization of the loop that contains them because the compiler cannot make assumptions about the state of registers over the call.

Automatic inlining of subprogram calls (using -inline=x,y...z, or -O4) is one way to let the compiler replace the actual call with the subprogram itself (pulling the subprogram into the loop). The subprogram source code for the routines that are to be inlined must be found in the same file as the calling routine.

There are other ways to eliminate subprogram calls:
- Use statement functions. If the external function being called is a simple math function, it might be possible to rewrite the function as a statement function or set of statement functions. Statement functions are compiled in-line and can be optimized.
- Push the loop into the subprogram. That is, rewrite the subprogram so that it can be called fewer times (outside the loop) and operate on a vector or array of values per call.

## Rationalizing Tangled Code

Complicated conditional operations within a computationally intensive loop can dramatically inhibit the compiler’s attempt at optimization. In general, a good rule to follow is to eliminate all arithmetic and logical IF’s, replacing them with block IF’s:

```
Original Code:
IF(A(I)-DELTA) 10,10,11
10  XA(I) = XB(I)*B(I,I)
   XY(I) = XA(I) - A(I)
   GOTO 13
11  XA(I) = Z(I)
   XY(I) = Z(I)
   IF(QZDATA.LT.0.) GOTO 12
   ICNT = ICNT + 1
   ROX(ICNT) = XA(I)-DELTA/2.
12  SUM = SUM + X(I)
13  SUM = SUM + XA(I)

Untangled Code:
IF(A(I).LE.DELTA) THEN
   XA(I) = XB(I)*B(I,I)
   XY(I) = XA(I) - A(I)
ELSE
   XA(I) = Z(I)
   XY(I) = Z(I)
   IF(QZDATA.GE.0.) THEN
      ICNT = ICNT + 1
      ROX(ICNT) = XA(I)-DELTA/2.
   ENDIF
   SUM = SUM + X(I)
ENDIF
SUM = SUM + XA(I)
```

Using block IF not only improves the opportunities for the compiler to generate optimal code, it also improves readability and assures portability.
Further Reading

The following reference books provide more details:

- *Analyzing Program Performance with Sun WorkShop*, Sun Microsystems, Inc.
SPARC: Parallelization

This chapter presents an overview of multiprocessor parallelization and describes the capabilities of Sun’s Fortran compilers. Implementation differences between f77 and f95 are noted.

Note – Fortran parallelization features require a Sun WorkShop HPC license.

Essential Concepts

Parallelizing (or multithreading) an application compiles the program to run on a multiprocessor system or in a multithreaded environment. Parallelization enables a single task, such as a DO loop, to run over multiple processors (or threads) with a potentially significant execution speedup.

Before an application program can be run efficiently on a multiprocessor system like the Ultra™ 60, Sun Enterprise™ Server 6500, or Sun Enterprise Server 10000, it needs to be multithreaded. That is, tasks that can be performed in parallel need to be identified and reprogrammed to distribute their computations across multiple processors or threads.

Multithreading an application can be done manually by making appropriate calls to the libthread primitives. However, a significant amount of analysis and reprogramming might be required. (See the Solaris Multithreaded Programming Guide for more information.)

Sun compilers can automatically generate multithreaded object code to run on multiprocessor systems. The Fortran compilers focus on DO loops as the primary language element supporting parallelism. Parallelization distributes the computational work of a loop over several processors without requiring modifications to the Fortran source program.
The choice of which loops to parallelize and how to distribute them can be left entirely up to the compiler (-autopar), specified explicitly by the programmer with source code directives (-explicitpar), or done in combination (-parallel).

**Note** – Programs that do their own (explicit) thread management should *not* be compiled with any of the compiler’s parallelization options. Explicit multithreading (calls to libthread primitives) cannot be combined with routines compiled with these parallelization options.

Not all loops in a program can be profitably parallelized. Loops containing only a small amount of computational work (compared to the overhead spent starting and synchronizing parallel tasks) may actually run more slowly when parallelized. Also, some loops cannot be safely parallelized at all; they would compute different results when run in parallel due to dependencies between statements or iterations.

Implicit loops (IF loops and Fortran 95 array syntax, for example) as well as explicit DO loops are candidates for automatic parallelization by the Fortran compilers.

Sun WorkShop compilers can detect loops that might be safely and profitably parallelized automatically. However, in most cases, the analysis is necessarily conservative, due to the concern for possible hidden side effects. (A display of which loops were and were not parallelized can be produced by the -loopinfo option.) By inserting source code directives before loops, you can explicitly influence the analysis, controlling how a specific loop is (or is not) to be parallelized. However, it then becomes your responsibility to ensure that such explicit parallelization of a loop does not lead to incorrect results.

Both `f77` and `f95` support two styles of explicit parallelization directives—Sun style and Cray style. In addition, `f95` supports the OpenMP 1.1 directives and runtime library routines. Explicit parallelization in Fortran is described on page 159.

**Speedups—What to Expect**

If you parallelize a program so that it runs over four processors, can you expect it to take (roughly) one fourth the time that it did with a single processor (a fourfold speedup)?

Probably not. It can be shown (by Amdahl’s law) that the overall speedup of a program is strictly limited by the fraction of the execution time spent in code running in parallel. This is true *no matter how many processors are applied*. In fact, if \( p \) is the percentage of the total program execution time that runs in parallel mode, the theoretical speedup limit is \( 100/(100-p) \); therefore, if only 60% of a program’s execution runs in parallel, the maximum increase in speed is 2.5, independent of the
number of processors. And with just four processors, the theoretical speedup for this
program (assuming maximum efficiency) would be just 1.8 and not 4. With
overhead, the actual speedup would be less.

As with any optimization, choice of loops is critical. Parallelizing loops that
participate only minimally in the total program execution time has only minimal
effect. To be effective, the loops that consume the major part of the runtime must be
parallelized. The first step, therefore, is to determine which loops are significant and
to start from there.

Problem size also plays an important role in determining the fraction of the program
running in parallel and consequently the speedup. Increasing the problem size
increases the amount of work done in loops. A triply nested loop could see a cubic
increase in work. If the outer loop in the nest is parallelized, a small increase in
problem size could contribute to a significant performance improvement (compared
to the unparallelized performance).

Steps to Parallelizing a Program

Here is a very general outline of the steps needed to parallelize an application:

1. **Optimize.** Use the appropriate set of compiler options to get the best serial
   performance on a single processor.

2. **Profile.** Using typical test data, determine the performance profile of the program.
   Identify the most significant loops.

3. **Benchmark.** Determine that the serial test results are accurate. Use these results
   and the performance profile as the benchmark.

4. **Parallelize.** Use a combination of options and directives to compile and build a
   parallelized executable.

5. **Verify.** Run the parallelized program on a single processor and single thread and
   check results to find instabilities and programming errors that might have crept in.
   (Set `$PARALLEL` or `SOMB_NUM_THREADS` to 1; see page 151)

6. **Test.** Make various runs on several processors to check results.

7. **Benchmark.** Make performance measurements with various numbers of processors
   on a dedicated system. Measure performance changes with changes in problem
   size (scalability).

8. **Repeat** steps 4 to 7. Make improvements to your parallelization scheme based on
   performance.
Data Dependency Issues

Not all loops are parallelizable. Running a loop in parallel over a number of processors usually results in iterations executing out of order. Moreover, the multiple processors executing the loop in parallel may interfere with each other whenever there are data dependencies in the loop.

Situations where data dependency issues arise include recurrence, reduction, indirect addressing, and data dependent loop iterations.

Recurrence

Variables that are set in one iteration of a loop and used in a subsequent iteration introduce cross-iteration dependencies, or recurrences. Recurrence in a loop requires that the iterations to be executed in the proper order. For example:

```fortran
  DO I=2,N
    A(I) = A(I-1) * B(I) + C(I)
  END DO
```

requires the value computed for A(I) in the previous iteration to be used (as A(I-1)) in the current iteration. To produce correct results, iteration I must complete before iteration I+1 can execute.

Reduction

Reduction operations reduce the elements of an array into a single value. For example, summing the elements of an array into a single variable involves updating that variable in each iteration:

```fortran
  DO K = 1,N
    SUM = SUM + A(I) * B(I)
  END DO
```

If each processor running this loop in parallel takes some subset of the iterations, the processors will interfere with each other, overwriting the value in SUM. For this to work, each processor must execute the summation one at a time, although the order is not significant.

Certain common reduction operations are recognized and handled as special cases by the compiler.
Indirect Addressing

Loop dependencies can result from stores into arrays that are indexed in the loop by subscripts whose values are not known. For example, indirect addressing could be order dependent if there are repeated values in the index array:

```
DO L = 1,NW
   A(ID(L)) = A(L) + B(L)
END DO
```

In the example, repeated values in ID cause elements in A to be overwritten. In the serial case, the last store is the final value. In the parallel case, the order is not determined. The values of A(L) that are used, old or updated, are order dependent.

Data Dependent Loops

You might be able to rewrite a loop to eliminate data dependencies, making it parallelizable. However, extensive restructuring could be needed.

Some general rules are:

- A loop is data independent only if all iterations write to distinct memory locations.
- Iterations may read from the same locations as long as no one iteration writes to them.

These are general conditions for parallelization. The compilers’ automatic parallelization analysis considers additional criteria when deciding whether to parallelize a loop. However, you can use directives to explicitly force loops to be parallelized, even loops that contain inhibitors and produce incorrect results.

Parallel Options and Directives Summary

The following table shows the Sun WorkShop 6 f77 and f95 compilation options related to parallelization.

**TABLE 10-1  Parallelization Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic (only)</td>
<td>-autopar</td>
</tr>
<tr>
<td>Automatic and Reduction</td>
<td>-autopar -reduction</td>
</tr>
<tr>
<td>Explicit (only)</td>
<td>-explicitpar</td>
</tr>
</tbody>
</table>
Notes on these options:
- -reduction requires -autopar.
- -autopar includes -depend and loop structure optimization.
- -parallel is equivalent to -autopar -explicitpar.
- -npatopar, -noexplicitpar, -noreduction are the negations.
- Parallelization options can be in any order, but they must be all lowercase.
- Reduction operations are not analyzed for explicitly parallelized loops.
- Use of any of the parallelization options requires a Sun WorkShop HPC license.
- -openmp is a macro for the combination of options:
  -mp=openmp -stackvar -explicitpar
- The options -loopinfo, -vpara, and -mp must be used in conjunction with one of the parallelization options -autopar, -explicitpar, or -parallel.

The following table summarizes the $f77$ and $f95$ Sun-style parallel directives.

<table>
<thead>
<tr>
<th>Parallel Directive</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>C$PAR TASKCOMMON</td>
<td>Declares a common block private to each thread</td>
</tr>
<tr>
<td>C$PAR DOALL  <em>optional qualifiers</em></td>
<td>Parallelizes next loop, if possible</td>
</tr>
<tr>
<td>C$PAR DOSERIAL</td>
<td>Inhibits parallelization of next loop</td>
</tr>
<tr>
<td>C$PAR DOSERIAL*</td>
<td>Inhibits parallelization of loop nest</td>
</tr>
</tbody>
</table>
Cray-style directives are similar (see page 176), but use a CMICS sentinel instead of C$PAR, and with different optional qualifiers on the DOALL directive. Use of these directives is explained in the section, “Explicit Parallelization” on page 159. Appendix E of the Fortran User’s Guide gives a detailed summary of all Fortran directives, including these and Fortran 95 OpenMP.

### Number of Threads

The PARALLEL (or OMP_NUM_THREADS) environment variable controls the maximum number of threads available to the program. Setting the environment variable tells the runtime system the maximum number of threads the program can use. The default is 1. In general, set the PARALLEL or OMP_NUM_THREADS variable to the available number of processors on the target platform.

The following example shows how to set it:

```plaintext
demo% setenv PARALLEL 4
C shell
-or-
demo$ PARALLEL=4
Bourne/Korn shell
demo$ export PARALLEL
```

In this example, setting PARALLEL to four enables the execution of a program using at most four threads. If the target machine has four processors available, the threads will map to independent processors. If there are fewer than four processors available, some threads could run on the same processor as others, possibly degrading performance.

The SunOS™ operating system command `psrinfo(1M)` displays a list of the processors available on a system:

```plaintext
demo% psrinfo
0  on-line  since 03/18/99 15:51:03
1  on-line  since 03/18/99 15:51:03
2  on-line  since 03/18/99 15:51:03
3  on-line  since 03/18/99 15:51:03
```
Stacks, Stack Sizes, and Parallelization

The executing program maintains a main memory stack for the initial thread executing the program, as well as distinct stacks for each helper thread. Stacks are temporary memory address spaces used to hold arguments and AUTOMATIC variables over subprogram invocations.

The default size of the main stack is about 8 megabytes. The Fortran compilers normally allocate local variables and arrays as STATIC (not on the stack). However, the -stackvar option forces the allocation of all local variables and arrays on the stack (as if they were AUTOMATIC variables). Use of -stackvar is recommended with parallelization because it improves the optimizer’s ability to parallelize subprogram calls in loops. -stackvar is required with explicitly parallelized loops containing subprogram calls. (See the discussion of -stackvar in the Fortran User’s Guide.)

Using the C shell (csh), the limit command displays the current main stack size as well as sets it:

```
demo% limit                      C shell example
  cputime       unlimited
  filesize      unlimited
  datasize      2097148 kbytes
  stacksize     8192 kbytes       <- current main stack size
  coredumpsize  0 kbytes
  descriptors   64
  memorysize    unlimited

demo% limit stacksize 65536      <- set main stack to 64Mb

demo% limit stacksize
  stacksize     65536 kbytes
```

With Bourne or Korn shells, the corresponding command is ulimit:

```
demo$ ulimit -a                  Korn Shell example
  time(seconds)       unlimited
  file(blocks)        unlimited
  data(kbytes)        2097148
  stack(kbytes)       8192
  coredump(blocks)    0
  nofiles(descriptors) 64
  vmemory(kbytes)     unlimited

demo$ ulimit -s 65536

demo$ ulimit -s 65536
```
Each helper thread of a multithreaded program has its own thread stack. This stack mimics the initial thread stack but is unique to the thread. The thread’s PRIVATE arrays and variables (local to the thread) are allocated on the thread stack. The default size is 2 Megabytes on SPARC V9 (UltraSPARC) platforms, 1 Megabyte otherwise. The size is set with the STACKSIZE environment variable:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo% setenv STACKSIZE 8192</td>
<td>Set thread stack size to 8 Mb  C shell</td>
</tr>
<tr>
<td>demo$ STACKSIZE=8192</td>
<td>-or-</td>
</tr>
<tr>
<td>demo$ export STACKSIZE</td>
<td>Bourne/Korn Shell</td>
</tr>
</tbody>
</table>

Setting the thread stack size to a value larger than the default may be necessary for some parallelized Fortran codes. However, it may not be possible to know just how large it should be, except by trial and error, especially if private/local arrays are involved. If the stack size is too small for a thread to run, the program will abort with a segmentation fault.

### Automatic Parallelization

With the -autopar and -parallel options, the f77 and f95 compilers automatically find DO loops that can be parallelized effectively. These loops are then transformed to distribute their iterations evenly over the available processors. The compiler generates the thread calls needed to make this happen.

### Loop Parallelization

The compiler’s dependency analysis transforms a DO loop into a parallelizable task. The compiler may restructure the loop to split out unparallelizable sections that will run serially. It then distributes the work evenly over the available processors. Each processor executes a different chunk of iterations.

For example, with four CPUs and a parallelized loop with 1000 iterations, each thread would execute a chunk of 250 iterations:

- Processor 1 executes iterations 1 through 250
- Processor 2 executes iterations 251 through 500
- Processor 3 executes iterations 501 through 750
- Processor 4 executes iterations 751 through 1000
Only loops that do not depend on the order in which the computations are performed can be successfully parallelized. The compiler’s dependence analysis rejects from parallelization those loops with inherent data dependencies. If it cannot fully determine the data flow in a loop, the compiler acts conservatively and does not parallelize. Also, it may choose not to parallelize a loop if it determines the performance gain does not justify the overhead.

Note that the compiler always chooses to parallelize loops using a static loop scheduling—simply dividing the work in the loop into equal blocks of iterations. Other scheduling schemes may be specified using explicit parallelization directives described later in this chapter.

Arrays, Scalars, and Pure Scalars

A few definitions, from the point of view of automatic parallelization, are needed:

- An array is a variable that is declared with at least one dimension.
- A scalar is a variable that is not an array.
- A pure scalar is a scalar variable that is not aliased—not referenced in an EQUIVALENCE or POINTER statement.

Example: Array/scalar:

```fortran
  dimension a(10)
  real m(100,10), s, u, x, z
  equivalence ( u, z )
  pointer ( px, x )
  s = 0.0
  ...
```

Both `m` and `a` are array variables; `s` is pure scalar. The variables `u`, `x`, `z`, and `px` are scalar variables, but not pure scalars.

Automatic Parallelization Criteria

DO loops that have no cross-iteration data dependencies are automatically parallelized by -autopar or -parallel. The general criteria for automatic parallelization are:

- Only explicit DO loops and implicit loops, such as IF loops and Fortran 95 array syntax are parallelization candidates.
- The values of array variables for each iteration of the loop must not depend on the values of array variables for any other iteration of the loop.
Calculations within the loop must not conditionally change any pure scalar variable that is referenced after the loop terminates.

Calculations within the loop must not change a scalar variable across iterations. This is called a loop-carried dependence.

The amount of work within the body of the loop must outweigh the overhead of parallelization.

£77: Apparent Dependencies

The £77 compiler may automatically eliminate a reference that appears to create a data dependency in the loop. One of the many such transformations makes use of private versions of some of the arrays. Typically, the compiler does this if it can determine that such arrays are used in the original loops only as temporary storage.

Example: Using -autopar, with dependencies eliminated by private arrays:

```fortran
parameter (n=1000)
real a(n), b(n), c(n,n)
do i = 1, 1000 <--Parallelized
   do k = 1, n
      a(k) = b(k) + 2.0
   end do
   do j = 1, n
      c(i,j) = a(j) + 2.3
   end do
end do
end
```

In the example, the outer loop is parallelized and run on independent processors. Although the inner loop references to array `a` appear to result in a data dependency, the compiler generates temporary private copies of the array to make the outer loop iterations independent.

Inhibitors to Automatic Parallelization

Under automatic parallelization, the compilers do not parallelize a loop if:

- The DO loop is nested inside another DO loop that is parallelized.
- Flow control allows jumping out of the DO loop.
- A user-level subprogram is invoked inside the loop.
- An I/O statement is in the loop.
- Calculations within the loop change an aliased scalar variable.
Nested Loops

In a multithreaded, multiprocessor environment, it is most effective to parallelize the outermost loop in a loop nest, rather than the innermost. Because parallel processing typically involves relatively large loop overhead, parallelizing the outermost loop minimizes the overhead and maximizes the work done for each thread. Under automatic parallelization, the compilers start their loop analysis from the outermost loop in a nest and work inward until a parallelizable loop is found. Once a loop within the nest is parallelized, loops contained within the parallel loop are passed over.

Automatic Parallelization With Reduction Operations

A computation that transforms an array into a scalar is called a reduction operation. Typical reduction operations are the sum or product of the elements of a vector. Reduction operations violate the criterion that calculations within a loop not change a scalar variable in a cumulative way across iterations.

Example: Reduction summation of the elements of a vector:

```fortran
s = 0.0
do i = 1, 1000
   s = s + v(i)
end do
t(k) = s
```

However, for some operations, if reduction is the only factor that prevents parallelization, it is still possible to parallelize the loop. Common reduction operations occur so frequently that the compilers are capable of recognizing and parallelizing them as special cases.

Recognition of reduction operations is not included in the automatic parallelization analysis unless the -reduction compiler option is specified along with -autopar or -parallel.

If a parallelizable loop contains one of the reduction operations listed in TABLE 10-3, the compiler will parallelize it if -reduction is specified.
Recognized Reduction Operations

The following table lists the reduction operations that are recognized by f77 and f95.

<table>
<thead>
<tr>
<th>Mathematical Operations</th>
<th>Fortran Statement Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>s = s + v(i)</td>
</tr>
<tr>
<td>Product</td>
<td>s = s * v(i)</td>
</tr>
<tr>
<td>Dot product</td>
<td>s = s + v(i) * u(i)</td>
</tr>
<tr>
<td>Minimum</td>
<td>s = amin( s, v(i))</td>
</tr>
<tr>
<td>Maximum</td>
<td>s = amax( s, v(i))</td>
</tr>
<tr>
<td>OR</td>
<td>do i = 1, n</td>
</tr>
<tr>
<td></td>
<td>b = b .or. v(i)</td>
</tr>
<tr>
<td></td>
<td>end do</td>
</tr>
<tr>
<td>AND</td>
<td>b = .true.</td>
</tr>
<tr>
<td></td>
<td>do i = 1, n</td>
</tr>
<tr>
<td></td>
<td>b = b .and. v(i)</td>
</tr>
<tr>
<td></td>
<td>end do</td>
</tr>
<tr>
<td>Count of non-zero elements</td>
<td>do i = 1, n</td>
</tr>
<tr>
<td></td>
<td>if(v(i).ne.0) k = k + 1</td>
</tr>
<tr>
<td></td>
<td>end do</td>
</tr>
</tbody>
</table>

All forms of the MIN and MAX function are recognized.

Numerical Accuracy and Reduction Operations

Floating-point sum or product reduction operations may be inaccurate due to the following conditions:

- The order in which the calculations are performed in parallel is not the same as when performed serially on a single processor.
- The order of calculation affects the sum or product of floating-point numbers. Hardware floating-point addition and multiplication are not associative. Roundoff, overflow, or underflow errors may result depending on how the operands associate. For example, (X*Y)*Z and X*(Y*Z) may not have the same numerical significance.

In some situations, the error may not be acceptable.
Example: Overflow and underflow, with and without reduction:

```fortran
demo% cat t3.f
 real A(10002), result, MAXFLOAT
 MAXFLOAT = r_max_normal()
 do 10 i = 1, 10000, 2
 A(i) = MAXFLOAT
 A(i+1) = -MAXFLOAT
10       continue
 A(5001)=-MAXFLOAT
 A(5002)=MAXFLOAT
 do 20 i = 1 ,10002     !Add up the array
   RESULT = RESULT + A(i)
20       continue
 write(6,*)(RESULT)
end

demo% setenv PARALLEL 2    (Number of processors is 2)
demo% f77 -silent -autopar t3.f
demo% a.out
  0.       {Without reduction, 0. is correct}
demo% f77 -silent -autopar -reduction t3.f
demo% a.out
  Inf      {With reduction, Inf. is not correct}
demo%
```

Example: Roundoff, get the sum of 100,000 random numbers between -1 and +1:

```fortran
demo% cat t4.f
 parameter ( n = 100000 )
 double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
 s = d_lcrans ( v, n, lb, ub ) ! Get n random nos. between -1 and +1
 s = 0.0
 do i = 1, n
   s = s + v(i)
 end do
 write(*, '(" s = ", e21.15)') s
end

demo% f77 -autopar -reduction t4.f
```
Results vary with the number of processors. The following table shows the sum of 100,000 random numbers between –1 and +1.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( s = 0.568582080884714E+02 )</td>
</tr>
<tr>
<td>2</td>
<td>( s = 0.568582080884722E+02 )</td>
</tr>
<tr>
<td>3</td>
<td>( s = 0.568582080884721E+02 )</td>
</tr>
<tr>
<td>4</td>
<td>( s = 0.568582080884724E+02 )</td>
</tr>
</tbody>
</table>

In this situation, roundoff error on the order of \(10^{-14}\) is acceptable for data that is random to begin with. For more information, see the Sun Numerical Computation Guide.

**Explicit Parallelization**

This section describes the source code directives recognized by `f77` and `f95` to explicitly indicate which loops to parallelize and what strategy to use.

The Sun WorkShop 6 Fortran compilers will accept both Sun-style and Cray-style parallelization directives to facilitate porting explicitly parallelized programs from other platforms.

The Fortran 95 compiler will also accept the OpenMP Fortran parallelization directives. The OpenMP Fortran specification is available at [http://www.openmp.org/](http://www.openmp.org/). The OpenMP directives, library routines, and environment variables are summarized in Appendix E of the Fortran User's Guide.

Explicit parallelization of a program requires prior analysis and deep understanding of the application code as well as the concepts of shared-memory parallelization.

DO loops are marked for parallelization by directives placed immediately before them. The compiler options `-parallel` or `-explicitpar` must be used for DO loops to be recognized and parallel code generated. Parallelization directives are comment lines that tell the compiler to parallelize (or not to parallelize) the `DO` loop that follows the directive. Directives are also called *pragmas*.

Take care when choosing which loops to mark for parallelization. The compiler generates threaded, parallel code for all loops marked with parallelization directives, even if there are data dependencies that will cause the loop to compute incorrect results when run in parallel.
If you do your own multithreaded coding using the 
libthread primitives, do not use any of the compilers’ parallelization options—the compilers cannot parallelize code that has already been parallelized with user calls to the threads library.

Parallelizable Loops

A loop is appropriate for explicit parallelization if:

■ It is a DO loop, but not a DO WHILE or Fortran 95 array syntax.
■ 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.
■ If the loop changes a scalar variable, that variable’s value is not used after the loop terminates. Such scalar variables are not guaranteed to have a defined value after the loop terminates, since the compiler does not automatically ensure a proper storeback for them.
■ For each iteration, any subprogram that is invoked inside the loop does not reference or change values of array variables for any other iteration.
■ The DO loop index must be an integer.

Scoping Rules: Private and Shared

A private variable or array is private to a single iteration of a loop. The value assigned to a private variable or array in one iteration is not propagated to any other iteration of the loop.

A shared variable or array is shared with all other iterations. The value assigned to a shared variable or array in an iteration is seen by other iterations of the loop.

If an explicitly parallelized loop contains shared references, then you must ensure that sharing does not cause correctness problems. The compiler does not synchronize on updates or accesses to shared variables.

If you specify a variable as private in one loop, and its only initialization is within some other loop, the value of that variable may be left undefined in the loop.

Subprogram Call in a Loop

A subprogram call in a loop (or in any subprograms called from within the called routine) may introduce data dependencies that could go unnoticed without a deep analysis of the data and control flow through the chain of calls. While it is best to parallelize outermost loops that do a significant amount of the work, these tend to be the very loops that involve subprogram calls.
Because such an interprocedural analysis is difficult and could greatly increase compilation time, automatic parallelization modes do not attempt it. With explicit parallelization, the compiler generates parallelized code for a loop marked with a DOALL directive even if it contains calls to subprograms. It is still the programmer’s responsibility to insure that no data dependencies exist within the loop and all that the loop encloses, including called subprograms.

Multiple invocations of a routine by different threads can cause problems resulting from references to local static variables that interfere with each other. Making all the local variables in a routine automatic rather than static prevents this. Each invocation of a subprogram then has its own unique store of local variables maintained on the stack, and no two invocations will interfere with each other.

Local subprogram variables can be made automatic variables that reside on the stack either by listing them on an AUTOMATIC statement or by compiling the subprogram with the -stackvar option. However, local variables initialized in DATA statements must be rewritten to be initialized in actual assignments.

**Note** – Allocating local variables to the stack can cause stack overflow. See “Stacks, Stack Sizes, and Parallelization” on page 152 about increasing the size of the stack.

**Inhibitors to Explicit Parallelization**

In general, the compiler parallelizes a loop if you explicitly direct it to. There are exceptions—some loops the compiler will not parallelize.

The following are the primary detectable inhibitors that might prevent explicitly parallelizing a DO loop:

- 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 that includes a call to a subroutine, then even if you request the compiler to parallelize loops in that subroutine, those loops are not run in parallel at runtime.

- 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.

By compiling with -vpara, you will get diagnostic messages when the compiler detects a problem while explicitly parallelizing a loop. The compiler may still parallelize the loop.
The following table lists typical parallelization problems detected by the compiler:

**TABLE 10-4  Explicit Parallelization Problems**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parallelized</th>
<th>Warning Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop is nested inside another loop that is parallelized.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Loop is in a subroutine called within the body of a parallelized loop.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Jumping out of loop is allowed by a flow control statement.</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Index variable of loop is subject to side effects.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Some variable in the loop has a loop-carried dependency.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I/O statement in the loop—usually unwise, because the order of the output is not predictable.</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Example: Nested loops:

```
...  
C$PAR DOALL
    do 900 i = 1, 1000  ! Parallelized (outer loop)
        do 200 j = 1, 1000  ! Not parallelized, no warning
    ...  
200     continue
900     continue
...  
```

Example: A parallelized loop in a subroutine:

```
program main
  ...
C$PAR DOALL
  do 100 i = 1, 200  
    ...  
    call calc (a, x)
    ...  
100    continue
...  
```

```
subroutine calc ( b, y )
  ...
C$PAR DOALL
  do 1 m = 1, 1000
    ...  
1    continue
    return
    end
```

Loop 100 runs in parallel.  
Loop 1 does not run in parallel.
In the example, the loop within the subroutine is not parallelized because the subroutine itself is run in parallel.

Example: Jumping out of a loop:

```
C$PAR DOALL
  do i = 1, 1000  ! ← Not parallelized, warning issued
    ...
    if (a(i) .gt. min_threshold ) go to 20
    ...
  end do
20      continue
  ...
```

Example: A variable in a loop has a loop-carried dependency:

```
C$PAR DOALL
  do 100 i = 1, 200  ! Parallelized, with warning
    y = y * i        ! y has a loop-carried dependency
    a(i) = y
  100      continue
  ...
```

I/O With Explicit Parallelization

You can do I/O in a loop that executes in parallel, provided that:

- It does not matter that the output from different threads is interleaved (program output is nondeterministic.)
- You can ensure the safety of executing the loop in parallel.
Example: I/O statement in loop

```
C$PAR DOALL
  do i = 1, 10  ! Parallelized with no warning (not advisable)
    k = i
    call show( k )
  end do
  subroutine show( j )
    write(6,1) j
  1    format('Line number ', i3, '.')
  end
  demo% f95 -explicitpar -vpara t13.f
  demo% setenv PARALLEL 2
  demo% a.out
  (The output displays the numbers 1 through 10, but in a non-deterministic order.)
```

Example: Recursive I/O:

```
do i = 1, 10  <-- Parallelized with no warning (unsafe)
  k = i
  print *, list( k )  <-- list is a function that does I/O
  end do
  end
  function list( j )
    write(6,"('Line number ', i3, '.')") j
    list = j
  end
  demo% f95 -mt t14.f
  demo% setenv PARALLEL 2
  demo% a.out
```

In the example, the program may deadlock in libF77_mt and hang. Press Control-C to regain keyboard control.

There are situations where the programmer might not be aware that I/O could take place within a parallelized loop. Consider a user-supplied exception handler that prints output when it catches an arithmetic exception (like divide by zero). If a parallelized loop provokes an exception, the implicit I/O from the handler may cause I/O deadlocks and a system hang.

In general:

- The library libF77_mt is MT safe, but mostly not MT hot.
- You cannot do recursive (nested) I/O if you compile with -mt.
As an informal definition, an interface is *MT safe* if:

- It can be simultaneously invoked by more than one thread of control.
- The caller is not required to do any explicit synchronization before calling the function.
- The interface is free of data races.

A *data race* occurs when the content of an address in memory is being updated by more than one thread, and that address is not protected by a lock. The value of that memory address is therefore nondeterministic—the two threads *race* to update the thread (but in this case, the one who gets there last, wins).

An interface is generally called *MT hot* if the implementation has been tuned for performance advantage, using the techniques of multithreading. See the Solaris *Multithreaded Programming Guide* for details.

### Sun-Style Parallelization Directives

Sun-style directives are enabled by default (or with the `-mp=sun` option) when compiling with the `-explicitpar` or `-parallel` options.

### Sun Parallelization Directives Syntax

A parallel directive consists of one or more *directive lines*. A Sun-style directive line is defined as follows:

```
C$PAR Directive [Qualifiers] <- Initial directive line
C$PAR& [More_Qualifiers] <- Optional continuation lines
```

- A directive line is case-insensitive.
- A directive line begins with a five-character sentinel: `C$PAR`, `*$PAR`, or `!$PAR`.
- With `f77` and `f95` fixed-format:
  - An *initial* directive line has a blank in column 6.
  - A *continuation* directive line has a nonblank in column 6.
  - Columns beyond 72 are ignored unless the `-e` option is specified.
- With `f95` free format:
  - Leading blanks are allowed before the sentinel.
  - The only sentinel recognized is `!$PAR`.
- Qualifiers, if any, follow directives—on the same line or continuation lines.
- Multiple qualifiers on one line are separated by commas.
- Spaces before, after, or within a directive or qualifier are ignored.
The Sun-style parallel directives are:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASKCOMMON</td>
<td>Declares variables in a COMMON block to be thread-private</td>
</tr>
<tr>
<td>DOALL</td>
<td>Parallelizes the next loop</td>
</tr>
<tr>
<td>DOSERIAL</td>
<td>Does not parallelize the next loop</td>
</tr>
<tr>
<td>DOSERIAL*</td>
<td>Does not parallelize the next nest of loops</td>
</tr>
</tbody>
</table>

Examples of Sun-style parallel directives:

```
C$PAR TASKCOMMON ALPHA
    COMMON /ALPHA/BZ, BY(100)
C$PAR DOALL
C$PAR DOSERIAL
C$PAR DOALL SHARED(I,K,X,V), PRIVATE(A)
    This one-line directive is equivalent to the three-line directive that follows.
C$PAR DOALL
    C$PAR SHARED(I,K,X,V)
    C$PAR PRIVATE(A)
```

**TASKCOMMON Directive**

The TASKCOMMON directive declares variables in a global COMMON block as *thread-private*: Every variable declared in a common block becomes a private variable to the thread, but remains global within the thread. Only named COMMON blocks can be declared TASKCOMMON.

The syntax of the directive is:

```
C$PAR TASKCOMMON common_block_name
```

The directive must appear immediately before or after every COMMON declaration for that named block.

This directive is effective only when compiled with `-explicitpar` or `-parallel`. Otherwise, the directive is ignored and the block is treated as a regular COMMON block.
Variables declared in TASKCOMMON blocks are treated as thread-private variables in all the DOALL loops and routines called from within the DOALL loops. Each thread gets its own copy of the COMMON block, so data written by one thread is not directly visible to other threads. During serial portions of the program, accesses are to the initial thread’s copy of the COMMON block.

Variables in TASKCOMMON blocks should not appear on any DOALL qualifiers, such as PRIVATE, SHARED, READONLY, and so on.

It is an error to declare a common block as task common in some but not all compilation units where the block is defined. A check at runtime for task common consistency can be enabled by compiling the program with the -xcommonchk=yes flag. (Enable the runtime check only during program development, as it can degrade performance.)

**DOALL Directive**

The DOALL directive requests the compiler to generate parallel code for the one DO loop immediately following it (if compiled with the -parallel or -explicitpar options).

**Note** – Analysis and transformation of reduction operations is not performed within explicitly parallelized loops.

Example: Explicit parallelization of a loop:

```plaintext
demo% cat t4.f
...
C$PAR DOALL
   do i = 1, n
      a(i) = b(i) * c(i)
   end do
   do k = 1, m
      x(k) = x(k) * z(k,k)
   end do
...
demo% f95 -explicitpar t4.f
```
DOALL Qualifiers

All qualifiers on the Sun-style DOALL directive are optional. The following table summarizes them:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Assertion</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVATE</td>
<td>Do not share variables (u1, \ldots) between iterations</td>
<td>DOALL PRIVATE((u1, u2, \ldots))</td>
</tr>
<tr>
<td>SHARED</td>
<td>Share variables (v1, v2, \ldots) between iterations</td>
<td>DOALL SHARED((v1, v2, \ldots))</td>
</tr>
<tr>
<td>MAXCPUS</td>
<td>Use no more than (n) CPUs (threads)</td>
<td>DOALL MAXCPUS((n))</td>
</tr>
<tr>
<td>READONLY</td>
<td>The listed variables are not modified in the DOALL loop</td>
<td>DOALL READONLY((v1, v2, \ldots))</td>
</tr>
<tr>
<td>STOREBACK</td>
<td>Save the last DO iteration values of variables (v1, \ldots)</td>
<td>DOALL STOREBACK((v1, v2, \ldots))</td>
</tr>
<tr>
<td>SAVELAST</td>
<td>Save the last DO iteration values of all private variables</td>
<td>DOALL SAVELAST</td>
</tr>
<tr>
<td>REDUCTION</td>
<td>Treat the variables (v1, v2, \ldots) as reduction variables.</td>
<td>DOALL REDUCTION((v1, v2, \ldots))</td>
</tr>
<tr>
<td>SCHEDTYPE</td>
<td>Set the scheduling type to (t).</td>
<td>DOALL SCHEDTYPE((t))</td>
</tr>
</tbody>
</table>

PRIVATE(varlist)

The PRIVATE(varlist) qualifier specifies that all scalars and arrays in the list varlist are private for the DOALL loop. Both arrays and scalars can be specified as private. In the case of an array, each thread of the DOALL loop gets a copy of the entire array. All other scalars and arrays referenced in the DOALL loop, but not contained in the private list, conform to their appropriate default scoping rules. (See page 160).

Example: Specify array a private in loop i:

```fortran
C$PAR DOALL PRIVATE(a)
do i = 1, n
  a(i) = b(i)
do j = 2, n
  a(j) = a(j-1) + b(j) * c(j)
end do
x(i) = f(a)
end do
```
**SHARE**D *(varlist)*

The **SHARE**D (varlist) qualifier specifies that all scalars and arrays in the list varlist are shared for the DOALL loop. Both arrays and scalars can be specified as shared. Shared scalars and arrays can be accessed in all the iterations of a DOALL loop. All other scalars and arrays referenced in the DOALL loop, but not contained in the shared list, conform to their appropriate default scoping rules.

Example: Specify a shared variable:

```c
C$PAR DOALL SHARED(y)
  do i = 1, n
    a(i) = y
  end do
```

In the example, the variable y has been specified as a variable whose value should be shared among the iterations of the i loop.

**READONLY** *(varlist)*

The **READONLY** (varlist) qualifier specifies that all scalars and arrays in the list varlist are read-only for the DOALL loop. Read-only scalars and arrays are a special class of shared scalars and arrays that are not modified in any iteration of the DOALL loop. Specifying scalars and arrays as **READONLY** indicates to the compiler that it does not need to use a separate copy of that scalar variable or array for each thread of the DOALL loop.

Example: Specify a read-only variable:

```c
x = 3
C$PAR DOALL SHARED(x),READONLY(x)
  do i = 1, n
    b(i) = x + 1
  end do
```

In the preceding example, x is a shared variable, but the compiler can rely on the fact that its value will not be modified in any iteration of the i loop because of its **READONLY** specification.
**STOREBACK (varlist)**

A STOREBACK scalar variable or array is one whose value is computed in a DOALL loop. The computed value can be used after the termination of the loop. In other words, the last loop iteration values of storeback scalars or arrays are visible after the DOALL loop.

Example: Specify the loop index variable as storeback:

```fortran
C$PAR DOALL PRIVATE(x), STOREBACK(x,i)
do i = 1, n
  x = ...
end do
... = i
... = x
```

In the preceding example, both the variables `x` and `i` are STOREBACK variables, even though both variables are private to the `i` loop. The value of `i` after the loop is `n+1`, while the value of `x` is whatever value it had at the end of the last iteration.

There are some potential problems for STOREBACK to be aware of.

The STOREBACK operation occurs at the last iteration of the explicitly parallelized loop, even if this is not the same iteration that last updates the value of the STOREBACK variable or array.

Example: STOREBACK variable potentially different from the serial version:

```fortran
C$PAR DOALL PRIVATE(x), STOREBACK(x)
do i = 1, n
  if (...) then
    x = ...
  end if
end do
print *,x
```

In the preceding example, the value of the STOREBACK variable `x` that is printed out might not be the same as that printed out by a serial version of the `i` loop. In the explicitly parallelized case, the processor that processes the last iteration of the `i` loop (when `i = n`) and performs the STOREBACK operation for `x`, might not be the same processor that currently contains the last updated value of `x`. The compiler issues a warning message about these potential problems.
SAVELAST

The SAVELAST qualifier specifies that all private scalars and arrays are STOREBACK variables for the DOALL loop.

Example: Specify SAVELAST:

```c
C$PAR DOALL PRIVATE(x,y), SAVELAST
   do i = 1, n
      x = ...  
      y = ...
   end do
   ... = i
   ... = x
   ... = y
C$PAR DOALL REDUCTION(x)
   do i = 1, n
      x = x + a(i)
   end do
```

In the example, variables x, y, and i are STOREBACK variables.

REDUCTION (varlist)

The REDUCTION(varlist) qualifier specifies that all variables in the list varlist are reduction variables for the DOALL loop. A reduction variable (or array) is one whose partial values can be individually computed on various processors, and whose final value can be computed from all its partial values.

The presence of a list of reduction variables requests the compiler to handle a DOALL loop as reduction loop by generating parallel reduction code for it.

Example: Specify a reduction variable:

```c
C$PAR DOALL REDUCTION(x)
   do i = 1, n
      x = x + a(i)
   end do
```

In the preceding example, the variable x is a (sum) reduction variable; the i loop is a (sum) reduction loop.
**SCHEDTYPE** (*t*)

**SCHEDTYPE** (*t*) specifies the scheduling type *t* be used to schedule the **DOALL** loop.

### TABLE 10-6  **DOALL SCHEDTYPE Qualifiers**

<table>
<thead>
<tr>
<th>Scheduling Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATIC</strong></td>
<td>Use <em>static scheduling</em> for this <strong>DO</strong> loop. (This is the default scheduling for Sun-style <strong>DOALL</strong> for both <strong>f77</strong> and <strong>f95</strong>.) Distribute all iterations uniformly to all available threads. Example: With 1000 iterations and 4 processors, each thread gets one chunk of 250 contiguous iterations.</td>
</tr>
<tr>
<td><strong>SELF [<strong>chunksize</strong>]</strong></td>
<td>Use <em>self-scheduling</em> for this <strong>DO</strong> loop. Each thread gets one chunk of <strong>chunksize</strong> iterations at a time, distributed in a nondeterministic order until all iterations are processed. Chunks of iterations may not be distributed uniformly to all available threads. • If <strong>chunksize</strong> is not provided, the compiler selects a value. Example: With 1000 iterations and <strong>chunksize</strong> of 4, each thread gets 4 iterations at a time until all iterations are processed.</td>
</tr>
<tr>
<td><strong>FACTORING [<strong>m</strong>]</strong></td>
<td>Use <em>factoring scheduling</em> for this <strong>DO</strong> loop. With <em>n</em> iterations initially and <em>k</em> threads, all the iterations are divided into groups of chunks of iterations, starting with the first group of <em>k</em> chunks of <em>n</em>/(2k) iterations each; the second group has <em>k</em> chunks of <em>n</em>/(4k) iterations, and so on. The <strong>chunksize</strong> for each group is the remaining iterations divided by 2k. Because <strong>FACTORING</strong> is dynamic, there is no guarantee that each thread gets exactly one chunk from each group. • At least <strong>m</strong> iterations must be assigned to each thread. • There can be one final smaller residual chunk. • If <strong>m</strong> is not provided, the compiler selects a value. Example: With 1000 iterations and <strong>FACTORING</strong>(3), and 4 threads, the first group has 4 chunks of 125 iterations each, the second has 4 chunks of 62 iterations each, the third group has 4 chunks of 31 iterations each, and so on.</td>
</tr>
<tr>
<td><strong>GSS [<strong>m</strong>]</strong></td>
<td>Use <em>guided self-scheduling</em> for this <strong>DO</strong> loop. With <em>n</em> iterations initially, and <em>k</em> threads, then: • Assign <em>n</em>/<em>k</em> iterations to the first thread. • Assign the remaining iterations divided by <em>k</em> to the second thread, and so on until all iterations have been processed. <strong>GSS</strong> is dynamic, so there is no guarantee that chunks of iterations are uniformly distributed to all available threads. • At least <strong>m</strong> iterations must be assigned to each thread. • There can be one final smaller residual chunk. • If <strong>m</strong> is not provided, the compiler selects a value. Example: With 1000 iterations and <strong>GSS</strong>(10), and 4 threads, distribute 250 iterations to the first thread, then 187 to the second thread, then 140 to the third thread, and so on.</td>
</tr>
</tbody>
</table>
Multiple Qualifiers

Qualifiers can appear multiple times with cumulative effect. In the case of conflicting qualifiers, the compiler issues a warning message, and the qualifier appearing last prevails.

Example: A three-line Sun-style directive (note conflicting MAXCPUS, SHARED, and PRIVATE qualifiers):

```
C$PAR DOALL MAXCPUS(4), READONLY(S), PRIVATE(A,B,X), MAXCPUS(2)
C$PAR DOALL SHARED(B,X,Y), PRIVATE(Y,Z)
C$PAR DOALL READONLY(T)
```

Example: A one-line equivalent of the preceding three lines:

```
C$PAR DOALL MAXCPUS(2), PRIVATE(A,Y,Z), SHARED(B,X), READONLY(S,T)
```

DOSERIAL Directive

The DOSERIAL directive disables parallelization of the specified loop. This directive applies to the one loop immediately following it.

Example: Exclude one loop from parallelization:

```
do i = 1, n
C$PAR DOSERIAL
  do j = 1, n
    do k = 1, n
      ...
    end do
  end do
end do
```

In the example, when compiling with \texttt{-parallel}, the \texttt{j} loop will not be parallelized by the compiler, but the \texttt{i} or \texttt{k} loop may be.

DOSERIAL* Directive

The DOSERIAL* directive disables parallelization of the specified nest of loops. This directive applies to the whole nest of loops immediately following it.
Example: Exclude a whole nest of loops from parallelization:

```fortran
  do i = 1, n
    C$PAR DOSERIAL*
      do j = 1, n
        do k = 1, n
          ...
        end do
      end do
    end do
  end do

  C$PAR DOSERIAL*
  do i = 1, 1000
    C$PAR DOALL
      do j = 1, 1000
        ...
      end do
    end do
  end do
```

In the example, when compiling with `-parallel`, the `j` and `k` loops will not be parallelized by the compiler, but the `i` loop may be.

**Interaction Between DOSERIAL* and DOALL**

If both DOSERIAL* and DOALL are specified for the same loop, the last one prevails.

Example: Specifying both DOSERIAL* and DOALL:

```fortran
  C$PAR DOSERIAL*
  do i = 1, 1000
    C$PAR DOALL
      do j = 1, 1000
        ...
      end do
    end do
  end do
```

In the example, the `i` loop is not parallelized, but the `j` loop is.

Also, the scope of the DOSERIAL* directive does not extend beyond the textual loop nest immediately following it. The directive is limited to the same function or subroutine that it appears in.
Example: **DO SERIAL** does not extend to a loop in a called subroutine:

```
program caller
    common /block/ a(10,10)
C$PAR DO SERIAL
    do i = 1, 10
        call callee(i)
    end do
end

subroutine callee(k)
    common /block/a(10,10)
    do j = 1, 10
        a(j,k) = j + k
    end do
    return
end
```

In the preceding example, **DO SERIAL** applies only to the $i$ loop and not to the $j$ loop, regardless of whether the call to the subroutine `callee` is inlined.

### Default Scoping Rules for Sun-Style Directives

For Sun-style (**C$PAR**) explicit directives, the compiler uses default rules to determine whether a scalar or array is shared or private. You can override the default rules to specify the attributes of scalars or arrays referenced inside a loop. (With Cray-style `!MIXS` directives, all variables that appear in the loop must be explicitly declared either shared or private on the `DOALL` directive.)

The compiler applies these default rules:

- All scalars are treated as `private`. A local copy of a scalar is made available for each thread executing the loop, and that local copy is used by that thread only.
- All array references are treated as `shared` references. Any write of an array element by one thread is visible to all threads. No synchronization is performed on accesses to shared variables.

If inter-iteration dependencies exist in a loop, then the execution may result in erroneous results. You must ensure that these cases do not arise. The compiler may sometimes be able to detect such a situation at compile time and issue a warning, but it does not disable parallelization of such loops.
Example: Potential problem through equivalence:

```fortran
  equivalence (a(1), y)
  C$PAR DOALL
    do i = 1, n
      y = i
      a(i) = y
    end do
```

In the example, since the scalar variable \( y \) has been equivalenced to \( a(1) \), we have a conflict with \( y \) as private and \( a(:) \) as shared by default, leading to possibly erroneous results when the parallelized \( i \) loop is executed. No diagnostic is issued in these situations.

You can fix the example by using \texttt{C$PAR DOALL PRIVATE(y)}.\texttt{.}

### Cray-Style Parallelization Directives

Parallel directives have two forms: Sun style and Cray style. The \texttt{f77} and \texttt{f95} default is Sun style (\texttt{-mp=sun}). To use Cray-style directives, you must compile with \texttt{-mp=cray}.

Mixing program units compiled with both Sun and Cray directives can produce incorrect results.

A major difference between Sun and Cray directives is that Cray style \textit{requires explicit scoping of every scalar and array in the loop as either SHARED or PRIVATE}.

The following table shows Cray-style directive syntax.

```fortran
!MIC$ DOALL
!MIC$  SHARED( v1, v2, ... )
!MIC$  PRIVATE( u1, u2, ... )
  ...optional qualifiers
```

### Cray Directive Syntax

A parallel directive consists of one or more \textit{directive lines}. A directive line is defined with the same syntax as Sun-style (page 165), except:

- The sentinels are \texttt{CMIC$}, \texttt{*MIC$}, or \texttt{!MIC$}, but only \texttt{!MIC$} is recognized with \texttt{f95} free-format.
Every variable or array referenced in the loop appears in a \texttt{SHARED} or \texttt{PRIVATE} qualifier.

The Cray directives are similar to Sun-style:

<table>
<thead>
<tr>
<th>Cray Directive</th>
<th>Compared With Sun-Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{DOALL}</td>
<td>different set of qualifiers and scheduling</td>
</tr>
<tr>
<td>\texttt{TASKCOMMON}</td>
<td>same as Sun-style</td>
</tr>
<tr>
<td>\texttt{DOSERIAL}</td>
<td>same as Sun-style</td>
</tr>
<tr>
<td>\texttt{DOSERIAL*}</td>
<td>same as Sun-style</td>
</tr>
</tbody>
</table>

\textbf{DOALL Qualifiers}

For Cray-style \texttt{DOALL}, the \texttt{PRIVATE} qualifier is required. Each variable within the \texttt{DO} loop must be qualified as private or shared, and the \texttt{DO} loop index must always be private. The following table summarizes available Cray-style qualifiers.

\textbf{TABLE 10-7 DOALL Qualifiers (Cray Style)}

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{SHARED( v1, v2, ... )}</td>
<td>Share the variables ( v1, v2, \ldots ) between iterations.</td>
</tr>
<tr>
<td>\texttt{PRIVATE( x1, x2, ... )}</td>
<td>Do not share the variables ( x1, x2, \ldots ) between iterations. That is, each thread has its own private copy of these variables.</td>
</tr>
<tr>
<td>\texttt{SAVELAST}</td>
<td>Save the last DO-iteration values of all \textit{private} variables in the loop.</td>
</tr>
<tr>
<td>\texttt{MAXCPUS( n )}</td>
<td>Use no more than ( n ) threads.</td>
</tr>
</tbody>
</table>
For Cray-style directives, the DOALL directive allows a single scheduling qualifier, for example, !MIC$& CHUNKSIZE(100). TABLE 10-8 shows the Cray-style DOALL directive scheduling qualifiers:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUIDED</td>
<td>Distribute the iterations by use of guided self-scheduling. This distribution minimizes synchronization overhead, with acceptable dynamic load balancing. The default chunk size is 64. GUIDED is equivalent to Sun-style GSS(64).</td>
</tr>
<tr>
<td>SINGLE</td>
<td>Distribute one iteration to each available thread. SINGLE is dynamic and equivalent to Sun-style SELF(1).</td>
</tr>
<tr>
<td>CHUNKSIZE( n )</td>
<td>Distribute n iterations to each available thread. n must be an integer expression. For best performance, n must be an integer constant. CHUNKSIZE(n) is equivalent to Sun-style SELF(n). Example: With 100 iterations and CHUNKSIZE(4), each thread gets 4 iterations at a time.</td>
</tr>
<tr>
<td>NUMCHUNKS( m )</td>
<td>If there are n iterations, distribute n/m iterations to each available thread. There can be one smaller residual chunk. m is an integer expression. For best performance, m must be an integer constant. NUMCHUNKS(m) is equivalent to Sun-style SELF(n/m) where n is the total number of iterations. Example: With 100 iterations and NUMCHUNKS(4), each thread gets 25 iterations at a time.</td>
</tr>
</tbody>
</table>

For both f77 and f95 the default scheduling type (when no scheduling type is specified on a Cray-style DOALL directive) is the Sun-style STATIC, for which there is no Cray-style equivalent.

Environment Variables

There are three environment variables used with parallelization:

- PARALLEL
- SUNW_MP THR_IDLE
- OMP_NUM_THREADS

(See also the STACKSIZE discussion on page 152)
PARALLEL and OMP_NUM_THREADS

To run a parallelized program in a multithreaded environment, you must set either the PARALLEL or OMP_NUM_THREADS environment variable prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set the PARALLEL or OMP_NUM_THREADS variable to the available number of processors on the target platform.

SUNW_MP_THR_IDLE

Use the SUNW_MP_THR_IDLE environment variable to control the end-of-task status of each thread executing the parallel part of a program. You can set the value of this variable to spin, sleep ns, or sleep nms. The default is spin, which means a thread spin-waits when it finishes its share of a parallel task until a new parallel task arrives. The other choice puts the thread to sleep after spin-waiting for n seconds (ns) or n milliseconds (nms). If a new task arrives before this wait-time the thread stops spinning and starts the new task.

In this example, at most four threads are created by the program. After finishing a parallel task, a thread spins for 50 ms. If within that time a new task has arrived for the thread, it executes it. Otherwise, the thread goes to sleep until a new task arrives.

Debugging Parallelized Programs

Debugging parallelized programs requires some extra effort. The following schemes suggest ways to approach this task.

First Steps at Debugging

There are some steps you can try immediately to determine the cause of errors.

- Turn off parallelization.

You can do one of the following:
Turn off the parallelization options—Verify that the program works correctly by compiling with -O3 or -O4, but without any parallelization.

Set the number of threads to one and compile with parallelization on—run the program with the environment variable PARALLEL set to 1.

If the problem disappears, then you can assume it was due to using multiple threads.

Check also for out of bounds array references by compiling with -C.

Problems using -autopar may indicate that the compiler is parallelizing something it should not.

Turn off -reduction.

If you are using the -reduction option, summation reduction may be occurring and yielding slightly different answers. Try running without this option.

Use the DO SERIAL directive to selectively disable automatic parallelization of individual loops.

Use fsplit.

If you have many subroutines in your program, use fsplit(1) to break them into separate files. Then compile some files with and without -parallel, and use f77 or f95 to link the .o files. You must specify -parallel on this link step. (See Fortran User's Guide section on consistent compiling and linking.)

Execute the binary and verify results.

Repeat this process until the problem is narrowed down to one subroutine.

Use -loopinfo.

Check which loops are being parallelized and which loops are not.

Use a dummy subroutine.

Create a dummy subroutine or function that does nothing. Put calls to this subroutine in a few of the loops that are being parallelized. Recompile and execute. Use -loopinfo to see which loops are being parallelized.

Continue this process until you start getting the correct results.

Use explicit parallelization.

Add the C$PAR DO ALL directive to a couple of the loops that are being parallelized. Compile with -explicitpar, then execute and verify the results. Use -loopinfo to see which loops are being parallelized. This method permits the addition of I/O statements to the parallelized loop.

Repeat this process until you find the loop that causes the wrong results.

Note: if you need -explicitpar only (without -autopar), do not compile with -explicitpar and -depend. This method is the same as compiling with -parallel, which, of course, includes -autopar.

Run loops backward serially.
Replace \( \text{DO } I=1,N \) with \( \text{DO } I=N,1,-1 \). Different results point to data dependencies.

- Avoid using the loop index.

\[
\begin{align*}
\text{Replace:} \\
& \quad \text{DO } I=1,N \\
& \quad \ldots \\
& \quad \text{CALL SNUBBER}(I) \\
& \quad \ldots \\
& \quad \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{With:} \\
& \quad \text{DO } I1=1,N \\
& \quad I=I1 \\
& \quad \ldots \\
& \quad \text{CALL SNUBBER}(I) \\
& \quad \ldots \\
& \quad \text{ENDDO}
\end{align*}
\]

**Debugging Parallel Code With dbx**

To use dbx on a parallel loop, temporarily rewrite the program as follows:

- Isolate the body of the loop in a file and subroutine of its own.
- In the original routine, replace loop body with a call to the new subroutine.
- Compile the new subroutine with \(-g\) and no parallelization options.
- Compile the changed original routine with parallelization and no \(-g\).
Example: Manually transform a loop to allow using \texttt{dbx} in parallel:

\begin{verbatim}
Original code:
demo\% \texttt{cat loop.f}
C\$PAR DOALL
  DO i = 1,10
    WRITE(0,\*) 'Iteration ', i
  END DO
END

Split into two parts: caller loop and loop body as a subroutine
demo\% \texttt{cat loop1.f}
C\$PAR DOALL
  DO i = 1,10
    k = i
    CALL loop_body ( k )
  END DO
END

demo\% \texttt{cat loop2.f}
SUBROUTINE loop_body ( k )
  WRITE(0,\*) 'Iteration ', k
RETURN
END

Compile caller loop with parallelization but no debugging
demo\% \texttt{f77 -O3 -c -explicitpar loop1.f}
Compile the subprogram with debugging but not parallelized
demo\% \texttt{f77 -c -g loop2.f}
Link together both parts into a.out
demo\% \texttt{f77 loop1.o loop2.o -explicitpar}
Ran a.out under dbx and put breakpoint into loop body subroutine
demo\% \texttt{dbx a.out} \hfill \text{\tiny \texttt{← Various dbx messages not shown}}
\begin{Verbatim}
(dbx) \texttt{stop in loop\_body}
(2) \texttt{stop in loop\_body}
(dbx) \texttt{run}
Running: a.out
(process id 28163)
dbx stops at breakpoint
(\@1 (\@1) stopped in loop\_body at line 2 in file
  "loop2.f"
  2 WRITE(0,\*) 'Iteration ', k
Now show value of k
(dbx) \texttt{print k}
  k = 1 \hfill \text{\tiny \texttt{← Various values other than 1 are possible}}
(dbx)
\end{Verbatim}
\end{verbatim}
This chapter treats issues regarding Fortran and C interoperability.

The discussion is inherently limited to the specifics of the Sun FORTRAN 77, Fortran 95, and C compilers.

Note – Material common to both FORTRAN 77 and Fortran 95 is presented in examples that use FORTRAN 77.

Compatibility Issues

Most C-Fortran interfaces must agree in all of these aspects:

- Function/subroutine: definition and call
- Data types: compatibility of types
- Arguments: passing by reference or value
- Arguments: order
- Procedure name: uppercase and lowercase and trailing underscore (_)
- Libraries: telling the linker to use Fortran libraries

Some C-Fortran interfaces must also agree on:

- Arrays: indexing and order
- File descriptors and stdio
- File permissions
Function or Subroutine?

The word function has different meanings in C and Fortran. Depending on the situation, the choice is important:

■ In C, all subprograms are functions; however, some may return a null (void) value.
■ In Fortran, a function passes a return value, but a subroutine does not.

When a Fortran routine 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.

When a C function calls a Fortran subprogram:

■ If the called Fortran subprogram is a function, call it from C as a function that returns a compatible data type.
■ If the called Fortran subprogram is a subroutine, call it from C as a function that returns a value of int (compatible to Fortran INTEGER*4) or void. A value is returned if the Fortran subroutine uses alternate returns, in which case it is the value of the expression on the RETURN statement. If no expression appears on the RETURN statement, and alternate returns are declared on SUBROUTINE statement, a zero is returned.

Data Type Compatibility

The tables below summarize the data sizes and default alignments for FORTRAN 77 and Fortran 95 data types. In both tables, note the following:

■ C data types int, long int, and long are equivalent (4 bytes). In a 64-bit environment and compiling with -xarch=v9 or v9a, long and pointers are 8 bytes. This is referred to as "LP64".
■ REAL*16 and COMPLEX*32, (REAL(KIND=16) and COMPLEX(KIND=16)), are available only on SPARC platforms. In a 64-bit environment and compiling with -xarch=v9 or v9a, alignment is on 16-byte boundaries.
■ Alignments marked 4/8 for SPARC indicate that alignment is 8-bytes by default, but on 4-byte boundaries in COMMON blocks. The maximum alignment in COMMON is 4-bytes.
■ The elements and fields of arrays and structures must be compatible.
■ You cannot pass arrays, character strings, or structures by value.
■ You can pass arguments by value from f77 to C, but not from C to f77, since %VAL() is not allowed in a Fortran dummy argument list.
FORTRAN 77 and C Data Types

TABLE 11-1 shows the sizes and allowable alignments for FORTRAN 77 data types. It assumes no compilation options affecting alignment or promoting default data sizes are applied. (See also the FORTRAN 77 Language Reference Manual).

<table>
<thead>
<tr>
<th>FORTRAN 77 Data Type</th>
<th>C Data Type</th>
<th>Size</th>
<th>Default Alignment SPARC</th>
<th>Default Alignment x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE X</td>
<td>char x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER X</td>
<td>unsigned char x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER*4 X</td>
<td>unsigned char x[n]</td>
<td>n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COMPLEX X</td>
<td>struct {float r,i;} x;</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*8 X</td>
<td>struct {float r,i;} x;</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE COMPLEX X</td>
<td>struct {double dr,di;} x;</td>
<td>16</td>
<td>4/8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*16 X</td>
<td>struct {double dr,di;} x;</td>
<td>16</td>
<td>4/8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*32 X</td>
<td>struct {long double dr,di;} x;</td>
<td>32</td>
<td>4/8/16</td>
<td>—</td>
</tr>
<tr>
<td>DOUBLE PRECISION X</td>
<td>double x</td>
<td>8</td>
<td>4/8</td>
<td>4</td>
</tr>
<tr>
<td>REAL X</td>
<td>float x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*4 X</td>
<td>float x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*8 X</td>
<td>double x</td>
<td>8</td>
<td>4/8</td>
<td>4</td>
</tr>
<tr>
<td>REAL*16 X</td>
<td>long double x</td>
<td>16</td>
<td>4/8/16</td>
<td>—</td>
</tr>
<tr>
<td>INTEGER X</td>
<td>int x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*2 X</td>
<td>short x</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER*4 X</td>
<td>int x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*8 X</td>
<td>long long int x</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL X</td>
<td>int x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*1 X</td>
<td>char x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL*2 X</td>
<td>short x</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL*4 X</td>
<td>int x</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*8 X</td>
<td>long long int x</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
SPARC: Fortran 95 and C Data Types

The following table similarly compares the Fortran 95 data types with C.

<table>
<thead>
<tr>
<th>Fortran 95 Data Type</th>
<th>C Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER x</td>
<td>unsigned char x;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER (LEN=n) x</td>
<td>unsigned char x[n];</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>COMPLEX x</td>
<td>struct {float r,i;} x;</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX (KIND=4) x</td>
<td>struct {float r,i;} x;</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX (KIND=8) x</td>
<td>struct {double dr,di;} x;</td>
<td>16</td>
<td>4/8</td>
</tr>
<tr>
<td>COMPLEX (KIND=16) x</td>
<td>struct {long double, dr,di;} x;</td>
<td>32</td>
<td>4/8/16</td>
</tr>
<tr>
<td>DOUBLE COMPLEX x</td>
<td>struct {double dr, di;} x;</td>
<td>16</td>
<td>4/8</td>
</tr>
<tr>
<td>DOUBLE PRECISION x</td>
<td>double x;</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>REAL x</td>
<td>float x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL (KIND=4) x</td>
<td>float x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL (KIND=8) x</td>
<td>double x;</td>
<td>8</td>
<td>4/8</td>
</tr>
<tr>
<td>REAL (KIND=16) x</td>
<td>long double x;</td>
<td>16</td>
<td>4/8/16</td>
</tr>
<tr>
<td>INTEGER x</td>
<td>int x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER (KIND=1) x</td>
<td>signed char x;</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER (KIND=2) x</td>
<td>short x;</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER (KIND=4) x</td>
<td>int x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER (KIND=8) x</td>
<td>long long int x;</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL x</td>
<td>int x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL (KIND=1) x</td>
<td>signed char x;</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL (KIND=2) x</td>
<td>short x;</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL (KIND=4) x</td>
<td>int x;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL (KIND=8) x</td>
<td>long long int x;</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Case Sensitivity

C and Fortran take opposite perspectives on case sensitivity:

- C is case sensitive—case matters.
- Fortran ignores case.

The `f77` and `f95` default is to ignore case by converting subprogram names to lowercase. It converts all uppercase letters to lowercase letters, except within character-string constants.
There are two usual solutions to the uppercase/lowercase problem:

- In the C subprogram, make the name of the C function all lowercase.
- Compile the Fortran program with the -U option, which tells the compiler to preserve existing uppercase/lowercase distinctions on function/subprogram names.

Use one of these two solutions, but not both.

Most examples in this chapter use all lowercase letters for the name in the C function, and do not use the f95 or f77 -U compiler option.

**Underscores in Routine Names**

The Fortran compiler normally appends an underscore (_) to the names of subprograms appearing both at entry point definition and in calls. This convention differs from C procedures or external variables with the same user-assigned name. All Fortran library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are three usual solutions to the underscore problem:

- In the C function, change the name of the function by appending an underscore to that name.
- Use the C() pragma to tell the Fortran compiler to omit those trailing underscores.
- Use the f77 and f95 -ext_names option to compile references to external names without underscores.

Use only one of these solutions.

The examples in this chapter could use the C() compiler pragma to avoid underscores. The C() pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C language, so the Fortran compiler does not append an underscore as it ordinarily does with external names. The C() directive for a particular function must appear before the first reference to that function. It must also appear in each subprogram that contains such a reference. The conventional usage is:

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

If you use this pragma, the C function does not need an underscore appended to the function name. (Pragma directives are described in the Fortran User’s Guide.)
Argument-Passing by Reference or Value

In general, Fortran routines pass arguments by reference. In a call, if you enclose an argument with the f77 and f95 nonstandard function %VAL(), the calling routine passes it by value.

In general, C passes arguments by value. If you precede an argument by the ampersand operator (&), C passes the argument by reference using a pointer. C always passes arrays and character strings by reference.

Argument Order

Except for arguments that are character strings, Fortran and C pass arguments in the same order. However, for every argument of character type, the Fortran routine passes an additional argument giving the length of the string. These are long int quantities in C, passed by value.

The order of arguments is:

■ Address for each argument (datum or function)
■ A long int for each character argument (the whole list of string lengths comes after the whole list of other arguments)

Example:

<table>
<thead>
<tr>
<th>This Fortran code fragment:</th>
<th>Is equivalent to this in C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER*7 S</td>
<td>char s[7];</td>
</tr>
<tr>
<td>INTEGER B(3)</td>
<td>int b[3];</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>CALL SAM( S, B(2) )</td>
<td>sam_( s, &amp;b[1], 7L );</td>
</tr>
</tbody>
</table>

Array Indexing and Order

Array indexing and order differ between Fortran and C.

Array Indexing

C arrays always start at zero, but by default Fortran arrays start at 1. There are two usual ways of approaching indexing.
You can use the Fortran default, as in the preceding 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 \( B(0) \) as follows:

```c
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: \( A(3, 2) \)

```
A(1, 1)  A(2, 1)  A(3, 1)  A(1, 2)  A(2, 2)  A(3, 2)
```

C arrays in row-major order: \( A[3][2] \)

```
```

For one-dimensional arrays, this is no problem. For two-dimensional and higher arrays, be aware of how subscripts appear and are used in all references and declarations—some adjustments might be necessary.

For example, it may be confusing to do part of a matrix manipulation in C and the rest in Fortran. It might be preferable to pass an *entire array* to a routine in the other language and perform *all* the matrix manipulation in that routine to avoid doing part in C and part in Fortran.

**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 do not have to recognize 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 in in the following table.

### TABLE 11-3  Comparing I/O Between Fortran and C

<table>
<thead>
<tr>
<th></th>
<th>Fortran Units</th>
<th>Standard I/O File Pointers</th>
<th>File Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Files Open</strong></td>
<td>Opened for reading and writing</td>
<td>Opened for reading; or Opened for writing; or Opened for both; or Opened for appending; See open(2)</td>
<td>Opened for reading; or Opened for writing; or Opened for both</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Formatted or unformatted</td>
<td>Always unformatted, but can be read or written with format-interpreting routines</td>
<td>Always unformatted</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>Direct or sequential</td>
<td>Direct access if the physical file representation is direct access, but can always be read sequentially</td>
<td>Direct access if the physical file representation is direct access, but can always be read sequentially</td>
</tr>
<tr>
<td><strong>Structure</strong></td>
<td>Record</td>
<td>Byte stream</td>
<td>Byte stream</td>
</tr>
<tr>
<td><strong>Form</strong></td>
<td>Arbitrary nonnegative integers from 0-2147483647</td>
<td>Pointers to structures in the user’s address space</td>
<td>Integers from 0-1023</td>
</tr>
</tbody>
</table>

### File Permissions

C programs typically open input files for reading and output files for writing or for reading and writing. A *f77* program can *OPEN* a file *READONLY* or with *READWRITE=*‘READ’ or ‘WRITE’ or ‘READWRITE’. *f95* supports the READWRITE specifier, but not READONLY.

Fortran tries to open a file with the maximum permissions possible, first for both reading and writing, then for each separately.

This event occurs transparently and is of concern only if you try to perform a READ, WRITE, or ENDFILE operation but you do not 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.
Libraries and Linking With the \texttt{f77} or \texttt{f95} Command

To link the proper Fortran and C libraries, use the \texttt{f77} or \texttt{f95} command to invoke the linker.

Example 1: Use \texttt{f77} to link:

```bash
demo% cc -c someCroutine.c
demo% f77 theF95routine.f someCroutine.o ← The linking step
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

Fortran Initialization Routines

Main programs compiled by \texttt{f77} and \texttt{f95} call dummy initialization routines \texttt{f77\_init} or \texttt{f90\_init} in the library at program start up. The routines in the library are dummies that do nothing. The calls the compilers generate pass pointers to the program’s arguments and environment. These calls provide software hooks the programmer can use to supply their own routines, in C, to initialize their program in any customized manner before the program starts up.

One possible use of these initialization routines to call \texttt{setlocale} for an internationalized Fortran program. Because \texttt{setlocale} does not work if \texttt{libc} is statically linked, only Fortran programs that are dynamically linked with \texttt{libc} should be internationalized.

The source code for the \texttt{init} routines in the library is

```c
void f77_init(int *argc_ptr, char ***argv_ptr, char ***envp_ptr) {} 
void f90_init(int *argc_ptr, char ***argv_ptr, Char ***envp_ptr) {}
```

The routine \texttt{f77\_init} is called by \texttt{f77} main programs. The routine \texttt{f90\_init} is called by \texttt{f95} main programs. The arguments are set to the address of \texttt{argc}, the address of \texttt{argv}, and the address of \texttt{envp}. 
Passing Data Arguments by Reference

The standard method for passing data between Fortran routines and C procedures is by reference. To a C procedure, a Fortran subroutine or function call looks like a procedure call with all arguments represented by pointers. The only peculiarity is the way Fortran handles character strings and functions as arguments and as the return value from a CHARACTER* function.

Simple Data Types

For simple data types (not COMPLEX or CHARACTER strings), define or pass each associated argument in the C routine as a pointer:

**TABLE 11-4**  Passing Simple Data Types

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer i</td>
<td>int i=100;</td>
</tr>
<tr>
<td>real r</td>
<td>float r;</td>
</tr>
<tr>
<td>external CSim</td>
<td>extern void fsim_(int *i, float *r);</td>
</tr>
<tr>
<td>i = 100</td>
<td>fsim_(&amp;i, &amp;r);</td>
</tr>
<tr>
<td>call CSim(i,r)</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>subroutine FSim(i,r)</td>
</tr>
<tr>
<td></td>
<td>integer i</td>
</tr>
<tr>
<td></td>
<td>real r</td>
</tr>
<tr>
<td></td>
<td>r = i</td>
</tr>
<tr>
<td></td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
COMPLEX Data

Pass a Fortran COMPLEX data item as a pointer to a C struct of two float or two double data types:

TABLE 11-5  Passing COMPLEX Data Types

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex w</td>
<td>struct cpx {float r, i;};</td>
</tr>
<tr>
<td>double complex z</td>
<td>struct cpx d1;</td>
</tr>
<tr>
<td>external CCmplx</td>
<td>struct cpx *w = &amp;d1;</td>
</tr>
<tr>
<td>call CCmplx(w,z)</td>
<td>struct dpx {double r, i;};</td>
</tr>
<tr>
<td>...</td>
<td>struct dpx d2;</td>
</tr>
<tr>
<td>struct cpx {float r, i;};</td>
<td>struct dpx *z = &amp;d2;</td>
</tr>
<tr>
<td>struct dpx {double r,i;};</td>
<td>fcmplx_({ w, z });</td>
</tr>
<tr>
<td>void complx_(</td>
<td>...</td>
</tr>
<tr>
<td>struct cpx *w,</td>
<td>subroutine FCmplx( w, z )</td>
</tr>
<tr>
<td>struct dpx *z)</td>
<td>complex w</td>
</tr>
<tr>
<td>{</td>
<td>double complex z</td>
</tr>
<tr>
<td>w -&gt; r = 32.;</td>
<td>w = (32., .007)</td>
</tr>
<tr>
<td>w -&gt; i = .007;</td>
<td>z = (66.67, 94.1)</td>
</tr>
<tr>
<td>z -&gt; r = 66.67;</td>
<td>return</td>
</tr>
<tr>
<td>z -&gt; i = 94.1;</td>
<td>end</td>
</tr>
</tbody>
</table>

In 64-bit environments and compiling with -xarch=v9, COMPLEX values are returned in registers.

Character Strings

Passing strings between C and Fortran routines is not recommended because there is no standard interface. However, note the following:

■ All C strings are passed by reference.

■ Fortran calls pass an additional argument for every argument with character type in the argument list. The extra argument gives the length of the string and is equivalent to a C long int passed by value. (This is implementation dependent.) The extra string-length arguments appear after the explicit arguments in the call.
A Fortran call with a character string argument is shown in the next example with its C equivalent:

<table>
<thead>
<tr>
<th>Fortran call:</th>
<th>C equivalent:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CHARACTER*7 S} \texttt{INTEGER B(3)} \texttt{CALL CSTRNG( S, B(2) )}</td>
<td>\texttt{char s[7]; int b[3]; cstrng_( s, &amp;b[1], 7L );}</td>
</tr>
</tbody>
</table>

If the length of the string is not needed in the called routine, the extra arguments may be ignored. However, note that Fortran does not automatically terminate strings with the explicit null character that C expects. This must be added by the calling program.

### One-Dimensional Arrays

Array subscripts in C start with 0.

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{integer i, Sum} \texttt{integer a(9)} \texttt{external FixVec} \texttt{call FixVec ( a, Sum )}</td>
<td>\texttt{extern void vecref_ ( int[], int * ); \ldots int i, sum; int v[9] = \ldots vecref_ ( v, &amp;sum ); \ldots}</td>
</tr>
</tbody>
</table>

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

```fortran
subroutine VecRef( v, total)
integer i, total, v(9)
total = 0
do i = 1,9
    total = total + v(i)
end do
...```

---

This concludes the extract from the Fortran Programming Guide document.
Two-Dimensional Arrays

Rows and columns between C and Fortran are switched.

**TABLE 11-8** Passing a Two-Dimensional Array

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL Q(10,20)</td>
<td>extern void qref_ ( int[][10], int *);</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Q(3,5) = 1.0</td>
<td>int m[20][10] = ... ;</td>
</tr>
<tr>
<td>CALL FIXQ(Q)</td>
<td>int sum;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>void fixq_( float a[20][10] )</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>qref_( m, &amp;sum );</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>a[5][3] = a[5][3] + 1.;</td>
<td>SUBROUTINE QREF(A,TOTAL)</td>
</tr>
<tr>
<td>...</td>
<td>INTEGER A(10,20), TOTAL</td>
</tr>
<tr>
<td>}</td>
<td>DO I = 1,10,10</td>
</tr>
<tr>
<td></td>
<td>DO J = 1,20,20</td>
</tr>
<tr>
<td></td>
<td>TOTAL = TOTAL + A(I,J)</td>
</tr>
<tr>
<td></td>
<td>END DO</td>
</tr>
<tr>
<td></td>
<td>END DO</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Structures

C and FORTRAN 77 structures and Fortran 95 derived types can be passed to each other’s routines as long as the corresponding elements are compatible.

**TABLE 11-9** Passing FORTRAN 77 STRUCTURE Records

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCTURE /POINT/</td>
<td></td>
</tr>
<tr>
<td>REAL X, Y, Z</td>
<td></td>
</tr>
<tr>
<td>END STRUCTURE</td>
<td></td>
</tr>
<tr>
<td>RECORD /POINT/ BASE</td>
<td></td>
</tr>
<tr>
<td>EXTERNAL FLIP</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>CALL FLIP( BASE )</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>struct point {</td>
<td></td>
</tr>
<tr>
<td>float x,y,z;</td>
<td></td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
<tr>
<td>void fflip_( struct point *v )</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>float t;</td>
<td></td>
</tr>
<tr>
<td>t = v -&gt; x;</td>
<td></td>
</tr>
<tr>
<td>v -&gt; x = v -&gt; y;</td>
<td></td>
</tr>
<tr>
<td>v -&gt; y = t;</td>
<td></td>
</tr>
<tr>
<td>v -&gt; z = -2.*(v -&gt; z);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| struct point { |
| float x,y,z; |
| };            |
| void fflip_ ( struct point *); |
| ...           |
| struct point d; |
| struct point *ptx = &d; |
| ...           |
| fflip_ (ptx); |
| ...           |

SUBROUTINE FFLIP(P)
| STRUCTURE /POINT/ |
| REAL X,Y,Z   |
| END STRUCTURE |
| RECORD /POINT/ P |
| REAL T       |
| T = P.X    |
| P.X = P.Y  |
| P.Y = T    |
| P.Z = -2.*P.Z |
| ...       |
### TABLE 11-10 Passing Fortran 95 Derived Types

<table>
<thead>
<tr>
<th>Fortran 95 calls C</th>
<th>C calls Fortran 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE point</td>
<td>struct point {</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>float x, y, z;</td>
</tr>
<tr>
<td>REAL :: : x, y, z</td>
<td>);</td>
</tr>
<tr>
<td>END TYPE point</td>
<td>extern void fflip_(</td>
</tr>
<tr>
<td>TYPE (point) base</td>
<td>struct point *);</td>
</tr>
<tr>
<td>EXTERNAL flip</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>struct point d;</td>
</tr>
<tr>
<td>CALL flip( base)</td>
<td>struct point *ptx = &amp;d;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>struct point {</td>
<td>fflip_(ptx);</td>
</tr>
<tr>
<td>float x, y, z;</td>
<td>...</td>
</tr>
<tr>
<td>};</td>
<td>SUBROUTINE FFLIP( P )</td>
</tr>
<tr>
<td>void fflip_( struct point *v )</td>
<td>TYPE POINT</td>
</tr>
<tr>
<td>{</td>
<td>REAL :: : X, Y, Z</td>
</tr>
<tr>
<td>float t;</td>
<td>END TYPE POINT</td>
</tr>
<tr>
<td>t = v -&gt; x;</td>
<td>TYPE (POINT) P</td>
</tr>
<tr>
<td>v -&gt; x = v -&gt; y;</td>
<td>REAL :: : T</td>
</tr>
<tr>
<td>v -&gt; y = t;</td>
<td>T = P%X</td>
</tr>
<tr>
<td>v -&gt; z = -2.*(v -&gt; z);</td>
<td>P%X = P%Y</td>
</tr>
<tr>
<td>}</td>
<td>P%Y = T</td>
</tr>
<tr>
<td></td>
<td>P%Z = -2.*P%Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Pointers

A FORTRAN 77 pointer can be passed to a C routine as a pointer to a pointer because the Fortran routine passes arguments by reference.

**TABLE 11-11** Passing a FORTRAN 77 POINTER

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL X</td>
<td>extern void fpass_( p2x );</td>
</tr>
<tr>
<td>POINTER (P2X, X)</td>
<td>...</td>
</tr>
<tr>
<td>EXTERNAL PASS</td>
<td>float *x;</td>
</tr>
<tr>
<td>P2X = MALLOC64(4)</td>
<td>float *p2x;</td>
</tr>
<tr>
<td>X = 0.</td>
<td>p2x = &amp;x;</td>
</tr>
<tr>
<td>CALL PASS(P2X)</td>
<td>fpass_(p2x);</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| void pass_(x)   | SUBROUTINE FPASS (P2X) |}

C pointers are compatible with Fortran 95 *scalar* pointers, but not *array* pointers.

**Passing Data Arguments by Value**

Call by value is available only for simple data with FORTRAN 77, and only by Fortran routines calling C routines. There is no way for a C routine to call a Fortran routine and pass arguments by value. It is not possible to pass arrays, character strings, or structures by value. These are best passed by reference.

Use the nonstandard Fortran function %VAL(arg) as an argument in the call.
In the following example, the Fortran routine passes x by value and y by reference. The C routine incremented both x and y, but only y is changed.

### TABLE 11-12 Passing Simple Data Arguments by Value: FORTRAN 77 Calling C

<table>
<thead>
<tr>
<th>Fortran 77 calls C</th>
<th>C-function <code>value_</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL x, y</td>
<td><code>void value_( float x, float *y)</code></td>
</tr>
<tr>
<td>x = 1.</td>
<td></td>
</tr>
<tr>
<td>y = 0.</td>
<td></td>
</tr>
<tr>
<td>PRINT *, x, y</td>
<td></td>
</tr>
<tr>
<td>CALL value( %VAL(x), y)</td>
<td></td>
</tr>
<tr>
<td>PRINT *, x, y</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>void value_( float x, float *y)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%f, %f\n&quot;, x, *y);</td>
<td></td>
</tr>
<tr>
<td>x = x + 1.;</td>
<td></td>
</tr>
<tr>
<td>*y = *y + 1.;</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%f, %f\n&quot;, x, *y);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Compiling and running produces output:</td>
<td></td>
</tr>
<tr>
<td>1.000000 0. x and y from Fortran</td>
<td></td>
</tr>
<tr>
<td>1.000000, 0.000000 x and y from C</td>
<td></td>
</tr>
<tr>
<td>2.000000, 1.000000 new x and y from C</td>
<td></td>
</tr>
<tr>
<td>1.00000 1.00000 new x and y from Fortran</td>
<td></td>
</tr>
</tbody>
</table>

---

### Functions That Return a Value

A Fortran function that returns a value of type BYTE (*FORTRAN 77 only*), INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or REAL*16 (*SPARC only*) is equivalent to a C function that returns a compatible type (see TABLE 11-1 and TABLE 11-2). There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.
Returning a Simple Data Type

The following example returns a REAL or float value. BYTE, INTEGER, LOGICAL, DOUBLE PRECISION, and REAL*16 are treated in a similar way:

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>real ADD1, R, S</td>
<td>float r, s;</td>
</tr>
<tr>
<td>external ADD1</td>
<td>extern float fadd1_();</td>
</tr>
<tr>
<td>R = 8.0</td>
<td>r = 8.0;</td>
</tr>
<tr>
<td>S = ADD1( R )</td>
<td>s = fadd1_( &amp;r );</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

-----------------------------

float add1_( pf )
float *pf;
{
    float f ;
    f = *pf;
    f++;
    return ( f );
}

-----------------------------

real function fadd1 (p)
real p
fadd1 = p + 1.0
return
end

Returning COMPLEX Data

A Fortran function returning COMPLEX or DOUBLE COMPLEX on SPARC V8 platforms is equivalent to a C function with an additional first argument that points to the return value in memory. The general pattern for the Fortran function and its corresponding C function is:

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>C function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX FUNCTION CF(a1, a2, ..., an)</td>
<td>cf_( return, a1, a2, ..., an)</td>
</tr>
<tr>
<td></td>
<td>struct ( float r, i ); *return</td>
</tr>
</tbody>
</table>
In 64-bit environments and compiling with \texttt{-xarch=v9}, COMPLEX values are returned in floating-point registers: COMPLEX and DOUBLE COMPLEX in \%f0 and \%f1, and COMPLEX*32 in \%f0, \%f1, \%f2, and \%f3. These registers are not directly accessible to C programs, preventing such interoperability between Fortran and C on SPARC V9 platforms for this case.

### Returning a CHARACTER String

Passing strings between C and Fortran routines is not encouraged. However, a Fortran character-string-valued function is equivalent to a C function with two additional first arguments—data address and string length. The general pattern for the Fortran function and its corresponding C function is:

<table>
<thead>
<tr>
<th>Fortran function</th>
<th>C function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER*(n) FUNCTION C((a1, \ldots, an))</td>
<td>void c_ (result, length, a1, \ldots, an)</td>
</tr>
<tr>
<td></td>
<td>char result[ ];</td>
</tr>
<tr>
<td></td>
<td>long length;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 11-14 Function Returning COMPLEX Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran calls C</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>COMPLEX U, V, RETCPX</td>
</tr>
<tr>
<td>EXTERNAL RETCPX</td>
</tr>
<tr>
<td>U = ( 7.0, -8.0)</td>
</tr>
<tr>
<td>V = RETCPX(U)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
</tbody>
</table>
Here is an example:

### TABLE 11-15  A Function Returning a CHARACTER String

<table>
<thead>
<tr>
<th>Fortran calls C</th>
<th>C calls Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER STRING<em>16, CSTR</em>9</td>
<td>void fstr_( char *, long,</td>
</tr>
<tr>
<td>STRING = ' '</td>
<td>char *, long *, long );</td>
</tr>
<tr>
<td>STRING = '123' // CSTR('*',9)</td>
<td>char sbf[9] = &quot;123456789&quot;;</td>
</tr>
<tr>
<td>...</td>
<td>char *p2rslt = sbf;</td>
</tr>
<tr>
<td>----------------------------</td>
<td>int rslt_len = sizeof(sbf);</td>
</tr>
<tr>
<td>void cstr_( char *p2rslt,</td>
<td>char ch = ' *';</td>
</tr>
<tr>
<td>long rslt_len,</td>
<td>int n = 4;</td>
</tr>
<tr>
<td>char *p2arg,</td>
<td>int ch_len = sizeof(ch);</td>
</tr>
<tr>
<td>long *p2n,</td>
<td>/* make n copies of ch in sbf</td>
</tr>
<tr>
<td>long arg_len )</td>
<td>*/</td>
</tr>
<tr>
<td>{ /* return n copies of arg */</td>
<td>fstr_( p2rslt, rslt_len,</td>
</tr>
<tr>
<td>int count, i;</td>
<td>&amp;ch, &amp;n, ch_len );</td>
</tr>
<tr>
<td>char *cp;</td>
<td>...</td>
</tr>
<tr>
<td>count = *p2n;</td>
<td>---------------------------</td>
</tr>
<tr>
<td>cp = p2rslt;</td>
<td>FUNCTION FSTR( C, N)</td>
</tr>
<tr>
<td>for (i=0; i&lt;count; i++)</td>
<td>CHARACTER FSTR*(*), C</td>
</tr>
<tr>
<td>{ *cp++ = *p2arg ;</td>
<td>FSTR = ' '</td>
</tr>
<tr>
<td>} }</td>
<td>DO I = 1,N</td>
</tr>
</tbody>
</table>

In this example, the C function and calling C routine must accommodate two initial extra arguments (a pointer to the result string and the length of the string) and one additional argument at the end of the list (length of character argument). Note that in the Fortran routine called from C, it is necessary to explicitly add a final null character.
Labeled COMMON

Fortran labeled COMMON can be emulated in C by using a global struct.

Table 11-16 Emulating Labeled COMMON

<table>
<thead>
<tr>
<th>Fortran COMMON Definition</th>
<th>C &quot;COMMON&quot; Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON /BLOCK/ ALPHA,NUM</td>
<td>extern struct block {</td>
</tr>
<tr>
<td>...</td>
<td>float alpha;</td>
</tr>
<tr>
<td></td>
<td>int num;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>extern struct block block_;</td>
</tr>
<tr>
<td></td>
<td>main ()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>block_.alpha = 32.;</td>
</tr>
<tr>
<td></td>
<td>block_.num += 1;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Note that the external name established by the C routine must end in an underscore to link with the block created by the Fortran program. Note also that the C directive #pragma pack may be needed to get the same padding as with Fortran. Both f77 and f95 align data in COMMON blocks to at most 4-byte boundaries.

Sharing I/O Between Fortran and C

Mixing Fortran I/O with C I/O (issuing I/O calls from both C and Fortran routines) is not recommended. It is better to do all Fortran I/O or all C I/O, not both.

The Fortran I/O library is implemented largely on top of the C standard I/O library. Every open unit in a Fortran program has an associated standard I/O file structure. For the stdin, stdout, and stderr streams, the file structure need not be explicitly referenced, so it is possible to share them.

If a Fortran main program calls C to do I/O, the Fortran I/O library must be initialized at program startup to connect units 0, 5, and 6 to stderr, stdin, and stdout, respectively. The C function must take the Fortran I/O environment into consideration to perform I/O on open file descriptors.
However, if a C main program calls a Fortran subprogram to do I/O, the automatic initialization of the Fortran I/O library to connect units 0, 5, and 6 to stderr, stdin, and stdout is lacking. This connection is normally made by a Fortran main program. If a Fortran function attempts to reference the stderr stream (unit 0) without the normal Fortran main program I/O initialization, output will be written to fort.0 instead of to the stderr stream.

The C main program can initialize Fortran I/O and establish the preconnection of units 0, 5, and 6 by calling the \texttt{f_init() \textsc{Fortran 77} library routine} at the start of the program and, optionally, \texttt{f_exit()} at termination.

Remember: even though the main program is in C, you should link with \texttt{f77}.

### Alternate Returns

Fortran’s alternate returns mechanism is obsolescent and should not be used if portability is an issue. There is no equivalent in C to alternate returns, so the only concern would be for a C routine calling a Fortran routine with alternate returns.

The Sun Fortran implementation returns the int value of the expression on the \texttt{RETURN} statement. This is implementation dependent and its use should be avoided.

<table>
<thead>
<tr>
<th>TABLE 11-17 Alternate Returns</th>
<th>Running the Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{int altret_ ( int * );}</td>
<td>\texttt{demo% cc -c tst.c}</td>
</tr>
<tr>
<td>\texttt{main ()}</td>
<td>\texttt{demo% f77 -o alt alt.f tst.o}</td>
</tr>
<tr>
<td>{}</td>
<td>\texttt{alt.f:}</td>
</tr>
<tr>
<td>\texttt{int k, m ;}</td>
<td>\texttt{altret:}</td>
</tr>
<tr>
<td>\texttt{k =0 ;}</td>
<td>\texttt{demo% alt}</td>
</tr>
<tr>
<td>\texttt{m = altret_ ( &amp;k ) ;}</td>
<td>1 2</td>
</tr>
<tr>
<td>\texttt{printf( &quot;%d %d\n&quot;, k, m);}</td>
<td>\texttt{The C routine receives the return value 2 from the Fortran routine because it executed the RETURN 2 statement.}</td>
</tr>
</tbody>
</table>
Index

SYMBOLS
Δ, blank space, 4

A
abrupt underflow, 85
agreement across routines, -Xlist, 63
aliasing, 115
align
data types, Fortran 95 vs. C, 186
data, Fortran 77 vs C, 185
errors across routines, -Xlist, 63
analyzing performance, 121
ar to create static library, 51, 55
arguments
reference versus value, C–Fortran interface, 188
array
differences between C and Fortran, 188
asa, Fortran print utility, 13
ASCII characters
maximum characters in data types, 111

B
-Bdynamic, -Bstatic options, 57
binary I/O, 28
bindings
POSIX, 60
static or dynamic (-B, -d), 57

C
C directive, 187
-C option, 77
C$PAR Sun-style directives, 165
call
in parallelized loops, 160
inhibiting optimization, 142
passing arguments by reference or value, 188
call graph profile, gprof, 125
call graphs, with -Xlistc option, 72
carriage-control, 109
case sensitivity, 186
C–Fortran interface
array indexing, 188
call arguments and ordering, 188
case sensitivity, 186
comparing I/O, 189
compatibility issues, 183
function compared to subroutine, 184
function names, 187, 192
passing data by value, 198, 199, 203
sharing I/O, 203
CHUNKSIZE directive qualifier, 178
CMIC$ Cray-style directives, 176
command line
passing runtime arguments, 23
redirection and piping, 25
command-line help, 17
common block
maps, -Xlist, 74
task common, 166
compile
  viewing source listing with diagnostics, 79
coverage analysis See tcov
cross reference table, -Xlist, 75

D
  -dalign option, 138
data
  alignment, Fortran 77 vs C, 185
  Hollerith, 111
  inspection, dbx, 78
  maximum characters in data types, 111
  representation, 111
  sizes, C vs. Fortran 77, 185
data dependency
  apparent, 155
  parallelization, 148
  restructuring to eliminate, 149
data race
  defined, 165
date, VMS, 107
dbx, 121
dd conversion utility, 32
debug, 63 to 79
  arguments, agree in number and type, 63
  common blocks, agree in size and type, 63
  compiler options, 77
  dbx, 78
  exceptions, 97
  index check of arrays, 77
  linker debugging aids, 45
  parameters, agree globally, 63
  segmentation fault, 77
  subscript array bounds checking, 77
  utilities, 13
declared but unused, checking, -Xlist, 64
denormalized number, 100
  -depend option, 139
diagnostics, source, 79
direct I/O, 27
  to internal files, 29
directives
  C() interface, 187
  OPT=n optimization levels, 138
  parallelization
  Cray, 176
  OpenMP, 159
  Sun, 159
  parallelization, summary, 150
display to terminal, -Xlist, 64
division by zero, 83
  -dn, -dy options, 57
  DOALL directive, 167
    qualifiers, 168
  DOSERIAL directive, 173
  DOSERIAL* directive, 173
dynamic libraries
    See libraries, dynamic
environment variables
  for parallelization, 178
  LD_LIBRARY_PATH, 48
  LOGICALNAMEMAPPING, 26
  OMP_NUM_THREADS, 151
  PARALLEL, 151
  passed to program, 23
  with IOINIT, 24
environment variables
  $SUN_PROFDATA, 130
equivalence block maps, -Xlist, 74
er_print command, 121
error
  messages
    listing with -XlistE, 72
    suppress with -Xlist, 72
    with error command, 13
  standard error
    accrued exceptions, 97
  establish a signal handler, 94
event management, dbx, 78
exceptions
  accrued, 89
  debugging, 97 to 99
  detecting, 94
  IEEE, 83
  ieee_handler, 91
  messages, 83
  suppressing warnings with ieee_flags, 83, 88
  trapping
    with -ftrap=mode option, 83
extensions and features, 12
external
  C functions, 187
  names, 186

F
f77_init, 191
f90_init, 191
FACTORING, directive qualifier, 172
-fast option, 137
features and extensions, 12
feedback, performance profiling, 138
file names
  on INCLUDE statements, 26
  passing to programs, 22
files
  internal, 29
  opening scratch files, 21
  passing file names to programs, 22
  permissions, C–Fortran interface, 190
  preconnected, 21
  standard error, 21
  standard input, 21
  standard output, 21
  tape, 32
fix and continue, dbx, 78
.fln files, 65
floating-point arithmetic, 81 to 111
  considerations, 100
  denormalized number, 100
  exceptions, 83
  IEEE, 82
  underflow, 100
  See also IEEE arithmetic, 82
-fns, disable underflow, 85
format
  edit descriptors, 109
Fortran
  features and extensions, 12
  libraries, 60
  utilities, 13
fpp command, 13
free format, 4
  -fsimple option, 139
fsplit, Fortran utility, 13
-ftrap=mode option, 83
function
  compared to subroutine, 184
  data type of, checking, -Xlist, 64
  names, Fortran vs. C, 186
  unused, checking, -Xlist, 64
  used as a subroutine, checking, -Xlist, 64
G
-G option, 59
GETARG library routine, 20, 23
GETC library routine, 33
GETENV library routine, 20, 23
global program checking
  strictness setting, 74
  See also -Xlist option
gprof command, 123
graphically monitor variables, dbx, 78
GSS, directive qualifier, 172
GUIDED, directive qualifier, 178
H
help, command-line, 17
Hollerith data, 111
I
IDATE VMS routine, 61
IEEE (Institute of Electronic and Electrical Engineers), 82
IEEE arithmetic
  754 standard, 82
  continue with wrong answer, 101
  exception handling, 84
  exceptions, 83
  excessive overflow, 102
  gradual underflow, 84, 100
  interfaces, 85
  signal handler, 94
  underflow handling, 84
  ieee_flags, 83, 85, 87
ieee_functions, 85
ieee_handler, 85, 91
ieee_retrospective, 83, 97
ieee_values, 85
INCLUDE, 26
include files
  list and cross checking with -XlistI, 73
inconsistency
  arguments, checking, -Xlist, 64
  named common blocks, checking, -Xlist, 64
indirect addressing
  data dependency, 149
inexact
  floating-point arithmetic, 83
information files, READMEs, 16
initialization, 191
inlining calls with -O4, 137
input/output, 19 to 34
  accessing files, 19
  binary, 28
  comparing Fortran and C I/O, 189
  dd conversion utility, 32
  direct I/O, 27
    to internal files, 29
  end-of-file on tape, 33
Fortran 95 considerations, 34
in parallelized loops, 163
inhibiting optimization, 142
inhibiting parallelization, 161
initialize for FORTRAN 77 from C, 204
internal I/O, 29
logical unit, 19
opening files, 21
preconnect units 0, 5, 6 from C, 204
preconnected units, 21
profiling, 131
random I/O, 27
redirection and piping, 25
scratch files, 21
tape, 31
  multifile, 33
interface
  problems, checking for, -Xlist, 64
internal files, 29
interval arithmetic, 14, 103
interval declaration, 103

IOINIT library routine, 24

L
-1x option, 49
labels, unused, -Xlist, 64
-Ldir option, 49
libF77, 60
libFposix, 60
libM77, 60
libraries, 43 to 62
dynamic
  creating, 55
  naming, 58
  position-independent code, 56
  specifying, 50
  tradeoffs, 56
in general, 43
linking, 44
load map, 44
math, 60
optimized, 141
POSIX, 61
provided with Sun WorkShop Fortran, 60
redistributable, 62
search order
  command line options, 49
  LD_LIBRARY_PATH, 48
  paths, 47
shared
  See dynamic
static
  creating, 51
  on SPARC V9, 58
ordering routines, 55
recompile and replace module, 55
tradeoffs, 51
Sun Performance Library, 14, 142
VMS, 60
libV77, 61
line width, output, -Xlist, 74
line-numbered listing, -Xlist, 65
linking
  binding options (-B, -d), 57
  consistent compile and link, 46
  libraries, 44
specifying static or dynamic, 57
mixing C and Fortran, 191
search order, 47
-lx, -Ldir, 49
troubleshooting errors, 50
lint-like checking across routines, -Xlist, 63
listing
cross-references with -Xlist, 75
line numbered with diagnostics, -Xlist, 63
-XlistL, 73
logical unit, 19
attached at runtime, 24
loop unrolling
and portability, 117
with -unroll, 139
-1V77 option, 61

M

-m linker option for load map, 45
macros
with make, 37
make, 35, 38
command, 37
macros, 37
makefile, 35
suffix rules, 38
makefile, 35
man pages, 14
MANPATH, path to man pages, 14
maps
common blocks, -Xlist, 74
equivalence blocks, -Xlist, 74
MAXCPUS, directive qualifier, 168, 177
measuring program performance See performance, profiling
monitor variables graphically, dbx, 78
multifile tapes, 33
multithreading
See parallelization

N
nonstandard_arithmetic(), 85
non-stopping I/O, 28
number of processors, 151
NUMCHUNKS directive qualifier, 178

O
OMP_NUM_THREADS, 151
OMP_NUM_THREADS environment variable, 179
OpenMP, 159
optimization See performance
options
dequiring, useful, 77
for optimization, 136 to 141
parallelization, 149
order of
linker libraries search, 47
linker search, 47
-lx, -Ldir options, 49
output
to terminal, -Xlist, 64
-Xlist report file, 73
overflow
excessive, 102
floating-point arithmetic, 83
locating, example, 99
with reduction operations, 157

P
PARALLEL environment variable, 151, 179
parallelization, 145 to 182
automatic, 153, 154
criteria, 154
CALL, loops with, 160
chunk distribution, 154
data dependency, 148
data race, 165
debugging, 179
definitions, 154
directives
Cray style, 176
OpenMP, 159
Sun style, 159
directives, summary, 150
environment variables, 178
explicit
criteria, 160
loop scheduling, 172
loop scheduling (Cray), 178
scoping rules, 160
scoping variables with Cray directives, 176
inhibitors
to automatic parallelization, 155
to explicit parallelization, 161
nested loops, 156
OpenMP, 159
options summary, 149
private and shared variables, 160
reduction operations, 156
specifying number of processors, 151
specifying stack sizes, 152
-stackvar option, 152
steps to, 147
what to expect, 146
performance
optimization, 135 to 144
choosing options, 135
further reading, 144
hand restructuring and portability, 115
inhibitors, 142
inlining calls, 137
libraries, 141
loop unrolling, 139
-0n options, 137
OPT=n directive, 138
specifying target hardware, 140
with runtime profile, 138
profiling, 121 to 133
gprof, 123
I/O, 131
overhead, 127
tcov, 127
time, 122
Sun Performance Library, 14
performance analyzer, 121
performance library, 142
-pic and -PIC options, 56
porting, 105 to 119
accessing files, 110
aliasing, 115
carriage-control, 109
data representation issues, 111
format edit descriptors, 109
Hollerith data, 111
initializing with Hollerith, 111
nonstandard coding, 114
obscure optimizations, 115
precision considerations, 110
strip-mining, 116
time functions, 105
troubleshooting guidelines, 118
uninitialized variables, 115
unrolled loops, 117
position-independent code
(-pic), 56
POSIX
bindings, libFposix, 60
Library, 61
pragma
See directives
preattached logical units, 24
preconnected units, 21
preprocessor, 13
preserve case, 186
preserving precision, 110
print, asa command, 13
PRIVATE, directive qualifier, 168, 177
process control, dbx, 78
processors (or threads), 151
program analysis, 63 to 79
program development tools, 35 to 42
make, 35
SCCS, 39
psrinfo SunOS command, 151
pure scalar variable
defined, 154
R
random I/O, 27
README files, 16
REASONLY, directive qualifier, 169
recurrence
data dependency, 148
reduction operations
data dependency, 148
numerical accuracy, 157
recognized by the compiler, 157
REDUCTION, directive qualifier, 171
referenced but not declared, checking, -Xlist, 64
retrospective summary of exceptions, 97
roundoff
  with reduction operations, 157
runtime
  arguments to program, 23

S
SAVELAST, directive qualifier, 171, 177
scalar
  defined, 154
SCCS
  checking in files, 42
  checking out files, 42
  creating files, 42
  creating SCCS directory, 40
  inserting keywords, 40
  putting files under SCCS, 39
SCHEDTYPE, directive qualifier, 172
scheduling, parallel loops, 172, 178
segmentation fault
  due to out-of-bounds subscripts, 77
SELF, directive qualifier, 172
shared library
  See libraries, dynamic, 55
SHARED, directive qualifier, 169, 177
sharing I/O, C–Fortran interface, 203
shippable libraries, 62
SIGFPE signal
  definition, 84, 91
  when generated, 94
SINGLE directive qualifier, 178
Solaris versions supported, 1
source
  diagnostics, 79
source code
  preprocessor, fpp, 13
  utility, fsplit, 13
source code control See SCCS
SPARC V9, 64-bit environments, 58
stack size and parallelization, 152
STACKSIZE environment variable, 153
  -stackvar option, 152
standard files
  error, 21
  input, 21
  output, 21
  redirection and piping, 25
standard_arithmetic(), 85
standards, 11
statement checking, -Xlist, 64
static libraries
  See libraries, static
STATIC, directive qualifier, 172
stdio, C–Fortran interface, 189
STOREBACK, directive qualifier, 170
strip-mining
  degrades portability, 116
subroutine
  compared to function, 184
  names, 186
  unused, checking, -Xlist, 64
  used as a function, checking, -Xlist, 64
suffix rules in make, 38
summing and reduction, automatic
  parallelization, 156
Sun Performance Library, 142
Sun WorkShop Performance Analyzer, 121
SUNW_MP_THR_IDLE environment variable, 179
supress
  unreferenced identifiers, -Xlist, 74
  warnings
    -Xlist, 74

T
tab format, 4
tape I/O, 31
  end-of-file, 33
  files, 32
  multilfile, 33
target
  specifying hardware, 140
task common, 166
TASKCOMMON directive, 166
tcov, 127
  and inlining, 128
  new style, -xprofile=tcov option, 130
  old style, -a option, 128
time command, 122
  multiprocessor interpretation, 122
time functions, 105
  summarized, 106
  VMS routines, 106
TIME VMS routine, 61
timing program execution, 122
TOPEN library routines, 31
transporting See porting
trapping
  exceptions with -ftrap=mode, 83
troubleshooting
  program fails, 119
  results not close enough, 118
type checking across routines, -Xlist, 63

U
  -U option, upper/lower case, 186
  undeclared variables, -u option, 77
underflow
  abrupt, 85
  floating-point arithmetic, 83
  gradual (IEEE), 84, 100
  simple, 100
  with reduction operations, 157
underscore, in external names, 187
uninitialized variables, 115
unit
  logical unit attached at runtime, 24
  preconnected units, 21
  -unroll option, 139
unused functions, subroutines, variables, labels,
  -Xlist, 64
uppercase, external names, 186
utilities, 13

V
  -V option, 78
%VAL(), pass by value, 188
variables
  aliased, 115
  private and shared, 160, 176
  undeclared, checking for with -u, 77
  uninitialized, 115
  unused, checking, -Xlist, 64
  used but unset, checking, -Xlist, 64
version checking, 78
VMS Fortran
  file names on INCLUDE, 26
  library libV77, 61
  time functions, 106

W
  watchpoints, dbx, 78
  width of output lines, -Xlist, 74

X
  -xl[d] option, 26
  -Xlist option, global program checking, 63 to 76
    .fln files directory, 72
    call graph, -Xlistc, 71
    cross reference, -XlistX, 71
    defaults, 65
    examples, 66, 75
    suboptions, 70 to 75
  -xmaxopt option, 138
  -xprofile option, 138
  -xtarget option, 140

Y
  Y2K (year 2000) considerations, 107

Z
  -ztext option, 59