Sun Studio 12: Fortran Programming Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–5262

070518@17466

Contents

Preface

The *Fortran Programming Guide* gives essential information about the the Sun™ Studio Fortran 95 compiler `f95`. It describes Fortran 95 input/output, program development, libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and interoperability.

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 Fortran compiler effectively. Familiarity with the Solaris™ operating environment or UNIX® in general is also assumed.

See also the companion *Fortran User's Guide* for information about the environment and command-line options for the `f95` compiler.

Typographic Conventions

TABLE P–1 Typeface Conventions

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

TABLE P–2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*n*`]`	`O4, O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*dir*`[:`*dir*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*f1*`[,…`*fn*`]`	`xinline=alpha,dos`

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

∇∇`36.001`

- The FORTRAN 77 standard used an older convention, spelling the name "FORTRAN" capitalized. 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 the library routine `GETENV` will appear as `getenv`(3F), implying that the man command to access this man page would be: `man -s 3F getenv`.

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell, Korn shell, and GNU Bourne-Again shell	$
Superuser for Bourne shell, Korn shell, and GNU Bourne-Again shell	#

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, these x86 related terms mean the following:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64' points out specific 64-bit information about AMD64 or EM64T systems.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms

  If your software is not installed in the `/opt` directory on a Solaris platform , ask your system administrator for the equivalent path on your system.
- Most manuals are available from the `docs.sun.com`[sm] web site. The following titles are available through your installed software on Solaris platforms only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*

  The release notes for both Solaris platforms and Linux platforms are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialog boxes, in the IDE.

The `docs.sun.com` web site (`http://www.docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: ■ *Standard C++ Library Class Reference* ■ *Standard C++ Library User's Guide* ■ *Tools.h++ Class Library Reference* ■ *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes	HTML on the Sun Developer Network portal at `http://developers.sun.com/sunstudio/documentation/ss12`
Man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms, and at `file:/opt/sun/sunstudio12/docs/index.html` on Linux platforms,
Online help	HTML available through the Help menu and Help buttons in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Sun Studio Documentation

For Solaris platforms, the following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` and `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Fortran Library Reference	Details the Fortran library and intrinsic routines
Fortran User's Guide	Describes the compile-time environment and command-line options for the `f95` compiler.
C User's Guide	Describes the compile-time environment and command-line options for the `cc` compiler.
C++ User's Guide	Describes the compile-time environment and command-line options for the `CC` compiler.
OpenMP API User's Guide	Summary of the OpenMP multiprocessing API, with specifics about the implementation.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating system.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Resources for Developers

Visit the Sun Developer Network Sun Studio portal at
`http://developers.sun.com/sunstudio` to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of the software, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

The Sun Studio portal is one of a number of additional resources for developers at the Sun Developer Network website, `http://developers.sun.com`.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL:

`http://www.sun.com/hwdocs/feedback`

Please include the part number of the document in the subject line of your email. For example, the part number for this document is 819-5262-10.

◆ ◆ ◆ CHAPTER 1

1 Introduction

The Sun™ Studio Fortran 95 compiler, **f95**, described here and in the companion *Fortran User's Guide,* is available under the Solaris™ operating environment on SPARC®, UltraSPARC®, and x64/x86 platforms. The compiler conforms to published Fortran language standards, and provides many extended features, including multiprocessor parallelization, sophisticated optimized code compilation, and mixed C/Fortran language support.

The **f95** compiler also provides a Fortran 77 compatibility mode that accepts most legacy Fortran 77 source codes. A separate Fortran 77 compiler is no longer included. See Chapter 5 of the *Fortran User's Guide* for information on FORTRAN 77 compatibility and migration issues.

1.1 Standards Conformance

- **f95** conforms to part one of the ISO/IEC 1539-1:1997 Fortran standards document.
- Floating-point arithmetic is based on IEEE standard 754-1985, and international standard IEC 60559:1989.
- **f95** provides support for the optimization-exploiting features of the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium Pro, and Xeon Intel® 64, on Solaris and Linux platforms.
- Sun Studio compilers conform to the OpenMP 2.5 shared memory parallelization API specifications. See the *OpenMP API User's Guide* for details.
- 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.

1.2 Features of the Fortran 95 Compiler

The Sun Studio Fortran 95 compiler provides the following features and extensions:

- Global program checking across routines for consistency of arguments, commons, parameters, and the like.
- Optimized automatic and explicit loop parallelization for multiprocessor systems.
- VAX/VMS Fortran extensions, including:
  - Structures, records, unions, maps
  - Recursion

  OpenMP 2.5 parallelization directives.
- 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 enabled Solaris environments on UltraSPARC and x64 platforms.
- Call-by-value using **%VAL**.
- Compatibility between Fortran 77 and Fortran 95 programs and object binaries.
- Interval Arithmetic programming.
- Some Fortran 2003 features, including Stream I/O.

See Appendix B of the *Fortran User's Guide* for details on new and extended features added to the compiler with each software release.

1.3 Other Fortran Utilities

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

- **Sun Studio 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).
- **fdumpmod**— A utility to display the names of modules contained in a file or archive. See **fdumpmod**(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)

1.4 Debugging Utilities

The following debugging utilities are available:

- **`-Xlist`**— A compiler option to check across routines for consistency of arguments, COMMON blocks, and so on.
- **Sun Studio `dbx`**—Provides a robust and feature-rich runtime and static debugger, and includes a performance data collector.

1.5 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, BLAS1, BLAS2, BLAS3, FFTPACK, VFFTPACK, and LINPACK generally available through Netlib (**`www.netlib.org`**).

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 accurate and can be used in a multiprocessing environment.

See the **`performance_library`** README file, and the *Sun Performance Library User's Guide* for details. (Man pages for the performance library routines are in section 3P.)

1.6 Interval Arithmetic

The Fortran 95 compiler provides the compiler flags **`-xia`** and **`-xinterval`** to enable new language extensions and generate the appropriate code to implement interval arithmetic computations. See the *Fortran 95 Interval Arithmetic Programming Reference* for details. (Interval arithmetic features are only supported on SPARC/UltraSPARC platforms.)

1.7 Man Pages

Online manual (**`man`**) pages provide immediate documentation about a command, function, subroutine, or collection of such things. The user's **`MANPATH`** environment variable should be set to the path of the installed Sun Studio **`man`** directory to access the Sun Studio man pages. Typically this is **`/opt/SUNWspro/man`** on Solaris.

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: **`f95`**(1) is accessed with **`man f95`**. Other sections, denoted by **`ieee_flags`**(3M) for example, are accessed using the **`-s`** option on the **`man`** command:

```
demo% man -s 3M ieee_flags
```

The Fortran library routines are documented in the man page section 3F.

The following lists **man** pages of interest to Fortran users:

f95(1)	The Fortran 95 command-line options
analyzer(1)	Sun Studio Performance Analyzer
asa(1)	Fortran carriage-control print output post-processor
dbx(1)	Command-line interactive debugger
fpp(1)	Fortran source code pre-processor
cpp(1)	C source code pre-processor
fdumpmod(1)	Display contents of a MODULE (**.mod**) file.
fsplit(1)	Pre-processor splits Fortran source routines into single files
ieee_flags(3M)	Examine, set, or clear floating-point exception bits
ieee_handler(3M)	Handle floating-point exceptions
matherr(3M)	Math library error handling routine
ild(1)	Incremental link editor for object files
ld(1)	Link editor for object files

1.8 README Files

The **README** pages on the Sun Developer Network (SDN) portal (`http://developers.sun.com/sunstudio`) describe new features, software incompatibilities, bugs, and information that was discovered after the manuals were printed. These README pages are part of the documentation on the portal for this release, and are also linked from the HTML documentation index that is part of the installed software at `file:/opt/SUNWspro/docs`.

TABLE 1–1 README Pages of Interest

README Page	Describes...
fortran_95	new and changed features, known limitations, documentation errata for this release of the Fortran 95 compiler, **f95**.
fpp_readme	overview of **fpp** features and capabilities

TABLE 1–1 README Pages of Interest *(Continued)*

README Page	Describes...
interval_arithmetic	overview of the interval arithmetic features in **f95**
math_libraries	optimized and specialized math libraries available.
profiling_tools	using the performance profiling tools, **prof**, **gprof**, and **tcov**.
runtime_libraries	libraries and executables that can be redistributed under the terms of the End User License.
performance_library	overview of the Sun Performance Library
openmp	new and changed features in the OpenMP parallelization API

The URL to the README page for each compiler is displayed by the **-xhelp=readme** command-line option. For example, the command:

```
% f95 -xhelp=readme
```

displays the URL for viewing the **fortran_95** README for this release on the SDN portal.

1.9 Command-Line Help

You can view very brief descriptions of the **f95** command line options by invoking the compiler's **-help** option as shown below:

```
%f95 -help=flags
Items within [ ] are optional. Items within < > are variable parameters.
Bar | indicates choice of literal values.
-someoption[={yes|no}] implies -someoption is equivalent to -someoption=yes

_____________________________________________________________
-a                 Collect data for tcov basic block profiling
-aligncommon[=<a>] Align common block elements to the specified boundary requirement; <a>={1|2|4|8|16}
-ansi              Report non-ANSI extensions.
-autopar           Enable automatic loop parallelization
-Bdynamic          Allow dynamic linking
-Bstatic           Require static linking
-C                 Enable runtime subscript range checking
-c                 Compile only; produce .o files but suppress
                   linking
...etc.
```

◆ ◆ ◆ CHAPTER 2

Fortran Input/Output

This chapter discusses the input/output features provided by the Sun Studio Fortran 95 compiler.

2.1 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.

2.1.1 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.

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.

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:

```
      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*(*), PREFIX*128
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)) //
     1                  NAME(2:LNBLNK(NAME))
      ELSE
            CALL GETCWD( PREFIX )
            FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     1                  '/' // NAME(:LNBLNK(NAME))
      ENDIF
      RETURN
      END
```

Compiling and running **GetFilNam.f** results in:

```
demo% pwd
/home/users/auser/subdir
demo% f95 -o getfil GetFilNam.f
demo% getfil
 ENTER FILE NAME:
getfil
 PATH IS: /home/users/auser/subdir/atest.f

demo%
```

These routines are further described in "2.1.4 Passing File Names to Programs" on page 24. See man page entries for **getarg**(3F), **getcwd**(3F), and **getenv**(3F) for details; these and other useful library routines are also described in the *Fortran Library Reference*.

2.1.2 Opening Files Without a Name

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

2.1.2.1 Opened as Scratch

Specifying **STATUS='SCRATCH'** in the **OPEN** statement opens a file with a name of the form **tmp.F***AAAxnnnnn*, where *nnnnn* is replaced by the current process ID, *AAA* is a string of three characters, and *x* is a letter; the *AAA* and *x* make the file name unique. This file is deleted upon termination of the program or execution of a **CLOSE** statement. When compiling in FORTRAN 77 compatibility mode (**-f77**), you can specify **STATUS='KEEP'** in the **CLOSE** statement to preserve the scratch file. (This is a non-standard extension.)

2.1.2.2 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.

2.1.2.3 Preconnected or Implicitly Named 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
- Standard output is logical unit 6
- Standard error is logical unit 0

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.***n*, where *n* is the logical unit number.

2.1.3 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.***n* is referenced, where *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:

```
demo% cat TestUnit.f
      IU=25
      WRITE( IU, '(I4)' ) IU
      END
demo%
```

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

```
demo% f95 -o testunit TestUnit.f
demo% testunit
demo% cat fort.25
  25
demo%
```

2.1.4 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.

2.1.4.1 Via Runtime Arguments and GETARG

The library routine **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 **OPEN** statement **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% f95 -o tstarg testarg.f
demo% tstarg AnyFileName
demo% cat AnyFileName
 Writing to file: AnyFileName
demo%
```

2.1.4.2 Via Environment Variables and GETENV

Similarly, the library routine **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% f95 -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 95 programs can use the intrinsic function **TRIM**, or the older FORTRAN 77 library routine **LNBLNK()**) 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.

2.1.4.3 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 2–1 **csh/sh/ksh** Redirection and Piping on the Command Line

Action	Using C Shell	Using Bourne or Korn Shell
Standard input— read from mydata	**myprog < mydata**	**myprog < mydata**
Standard output— write (overwrite) myoutput	**myprog > myoutput**	**myprog > myoutput**
Standard output— write/append to myoutput	**myprog >> myoutput**	**myprog >> myoutput**

TABLE 2–1 **csh/sh/ksh** Redirection and Piping on the Command Line *(Continued)*

Action	Using C Shell	Using Bourne or Korn Shell
Redirect standard error to a file	**myprog >& errorfile**	**myprog 2> errorfile**
Pipe standard output to input of another program	**myprog1 | myprog2**	**myprog1 | myprog2**
Pipe standard error and output to another program	**myprog1 |& myprog2**	**myprog1 2>&1 | myprog2**

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

2.2 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=***n*, 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*:

```
      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*:

```
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.

2.3 Binary I/O

Sun Studio Fortran 95 extends 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.

2.4 Stream I/O

A new "stream" I/O scheme of the Fortran 2003 standard is implemented in **f95**. Stream I/O access treats a data file as a continuous sequence of bytes, addressable by a positive integer starting from 1. Declare a stream I/O file with the **ACCESS='STREAM'** specifier on the **OPEN** statement. File positioning to a byte address requires a **POS=***scalar_integer_expression* specifier on a **READ** or **WRITE** statement. The **INQUIRE** statement accepts **ACCESS='STREAM'**, a specifier **STREAM=***scalar_character_variable*, and **POS=***scalar_integer_variable*.

Stream I/O is very useful when interoperating with files created or read by C programs, as is shown in the following example:

Fortran 95 program reads files created by C `fwrite()`

```
program reader
 integer:: a(1024), i, result
 open(file="test", unit=8, access="stream",form="unformatted")
! read all of a
 read(8) a
 do i = 1,1024
   if (a(i) .ne. i-1) print *,'error at ', i
 enddo
! read the file backward
 do i = 1024,1,-1
   read(8, pos=(i-1)*4+1) result
   if (result .ne. i-1) print *,'error at ', i
 enddo
 close(8)
end
```

C program writes to a file

```
#include <stdio.h>
int binary_data[1024];

/* Create a file with 1024 32-bit integers */
int
main(void)
{
    int i;
    FILE *fp;

    for (i = 0; i < 1024; ++i)
        binary_data[i] = i;
    fp = fopen("test", "w");
    fwrite(binary_data, sizeof(binary_data), 1, fp);
    fclose(fp);
}
```

The C program writes 1024 32-bit integers to a file using C **fwrite()**. The Fortran 95 reader reads them once as an array, and then reads them individually going backwards through the file. The **pos=** specifier in the second **read** statement illustrates that positions are in bytes, starting from byte 1 (as opposed to C, where they start from byte 0).

2.5 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.
- Direct I/O on internal files. (The Fortran 95 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. This non-standard extension is only available in FORTRAN 77 compatibility mode by compiling with the **-f77** flag.
- 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):

```
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% f95 -o tstintern intern1.f
demo% tstintern
 12 99
 12 99
demo%
```

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

```
demo% cat intern2.f
      CHARACTER  LINE(4)*16
      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% f95 intern2.f
demo% a.out
   81  81  82  82  83  83
demo%
```

Example: Direct access read from an internal file (one record), in **-f77** compatibility mode:

```
demo% cat intern3.f
      CHARACTER LINE(4)*16
      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
demo% f95 -f77 intern3.f
demo% a.out
   83  83
demo%
```

2.6 Binary I/O Between Big-Endian and Little-Endian Platforms

A new compiler flag **-xfilebyteorder** provides support for binary I/O files when moving between SPARC and x86 platforms. The flag identifies the byte-order and byte-alignment of unformatted I/O files.

For example,

```
-xfilebyteorder=little4:%all,big16:20
```

would specify that all files (except those opened as "**SCRATCH**") contain "little-endian" data aligned on 4-byte boundaries (for example 32-bit x86), except for Fortran unit 20, which is a 64-bit "big-endian" file (for example 64-bit SPARC).

For details, see the **f95**(1) man page or the *Fortran User's Guide*.

2.7 Legacy I/O Considerations

Fortran 95 and legacy Fortran 77 programs 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:

```
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 were by **f77**.

See the *Fortran User's Guide* for additional information about Fortran 95 I/O extensions and compatibility between **f95** and **f77**.

◆ ◆ ◆ CHAPTER 3

Program Development

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

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.

3.1 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**(1S).

3.1.1 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
demo%
```

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
      f95 pattern.o computepts.o startupcore.o -lcore95 \
      -lcore -lsunwindow -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 -c -u pattern.f
computepts.o: computepts.f commonblock
      f95 -c -u computepts.f
startupcore.o: startupcore.f
      f95 -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.

3.1.2 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.

3.1.3 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 **f95** link command for target **pattern** in **makefile**:

```
pattern: $(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
```

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

3.1.4 Overriding Macro Values

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

```
FFLAGS=–u
OBJ = pattern.o computepts.o startupcore.o
pattern: $(OBJ)
```

```
      f95 $(FFLAGS) $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o:
      f95 $(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:

```
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 **f95**. Note that **"FFLAGS="** can also be used on the command to reset the macro to a null string so that it has no effect.

3.1.5 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.

The default rules are in the file **/usr/share/lib/make/make.rules**. When recognizing default suffix rules, **make** passes as arguments any flags specified by the **FFLAGS** macro, the **-c** flag, and the name of the source file to be compiled. Also, the **make.rules** file uses the name assigned by the **FC** macro as the name of the Fortran compiler to be used.

The example below demonstrates this rule twice:

```
FC = f95
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=–u
pattern: $(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

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

There are default suffix rules for **.f90** files that will invoke the **f95** compiler.

However, unless you define the **FC** macro to be **f95,** the default suffix rules for **.f** and **.F** files call **f77** and not **f95**.

Furthermore, there are no suffix rules currently defined for **.f95** and **.F95** files, and **.mod** Fortran 95 module files will invoke the Modula compiler. To remedy this requires creating your own local copy of the **make.rules** file in the directory in which make is called, and modifying the file to add **.f95** and **.F95** suffix rules, and delete the suffix rules for **.mod**. See the **make**(1S) man page for details.

3.1.6 .KEEP_STATE and Special Dependency Checking

Use the special target **.KEEP_STATE** to check for command dependencies and hidden dependencies.

When the **.KEEP_STATE:** target is effective, **make** checks the command for building a target against the state file. If the command has changed since the last **make** run, **make** rebuilds the target.

When the **.KEEP_STATE:** target is effective, **make** reads reports from **cpp**(1) and other compilation processors for any "hidden" files, such as **#include** files. If the target is out of date with respect to any of these files, **make** rebuilds it.

3.2 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**.

3.2.1 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

3.2.1.1 Creating 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.

3.2.1.2 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% f95 -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.

3.2.1.3 Creating SCCS Files

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

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

3.2.2 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:

```
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:

```
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.

◆ ◆ ◆ CHAPTER 4

Libraries

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

4.1 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 **lib***name***.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 **lib***name***.so**. The **.so** suffix refers to *shared object*.

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

- Fortran 95 libraries: **libfsu**, **libfui**, **libfai**, **libfai2**, **libfsumai**, **libfprodai**, **libfminlai**, **libfmaxlai**, **libminvai**, **libmaxvai**, **libifai**, **libf77compat**
- 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.

4.2 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.

```
demo% setenv LD_OPTIONS '–m -Dfiles'
demo% f95 -o myprog myprog.f
Example: Using LD_OPTIONS to create a load map:
```

Some linker options do have compiler command-line equivalents that can appear directly on the **f95** command. These include **-B***x*, **-d***x*, **-G**, **-h***name*, **-R***path*, and **-ztext**. See the **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*.

4.2.1 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 **–m** to generate a load map:

```
demo% setenv LD_OPTIONS '-m'
demo% f95 any.f
any.f:
 MAIN:
            LINK EDITOR MEMORY MAP

output    input    virtual
section   section  address         size

.interp             100d4          11
            .interp 100d4          11 (null)
```

```
.hash               100e8          2e8
            .hash    100e8        2e8 (null)
.dynsym             103d0          650
            .dynsym 103d0          650 (null)
.dynstr             10a20          366
            .dynstr 10a20          366 (null)
.text               10c90          1e70
.text               10c90    00 /opt/SUNWspro/lib/crti.o
.text               10c90    f4 /opt/SUNWspro/lib/crt1.o
.text               10d84    00 /opt/SUNWspro/lib/values-xi.o
.text               10d88    d20 sparse.o
...
```

4.2.2 Listing Other Information

Additional linker debugging features are available through the linker's **-D***keyword* option. A complete list can be displayed using **-Dhelp**.

Example: List linker debugging aid options using the **-Dhelp** option:

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

For example, the **-Dfiles** linker option lists all the files and libraries referenced during the link process:

```
demo% setenv LD_OPTIONS '-Dfiles'
demo% f95 direct.f
direct.f:
 MAIN direct:
debug: file=/opt/SUNWspro/lib/crti.o  [ ET_REL ]
debug: file=/opt/SUNWspro/lib/crt1.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 ]
      ...
```

See the *Linker and Libraries Guide* for further information on these linker options.

4.2.3 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 some options requires linking with the same options. Also, a number of options require that *all* source files be compiled with that option, *including* the link step.

The option descriptions in the *Fortran User's Guide* identify such options.

Example: Compiling **sbr.f** with **-fast**, compiling a C routine, and then linking in a separate step:

```
demo% f95 -c -fast sbr.f
demo% cc -c -fast simm.c
demo% f95 -fast sbr.o simm.o          link step; passes  -fast  to the linker
```

4.3 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**.

4.3.1 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. A standard install puts the Sun Studio compiler software under **/opt/SUNWspro/**.

4.3.1.1 Static Linking

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

/opt/SUNWspro/lib	Sun Studio shared libraries
/usr/ccs/lib/	Standard location for SVr4 software
/usr/lib	Standard location for UNIX software

These are the *default* paths used by the linker.

4.3.1.2 Dynamic Linking

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

- Paths specified by user with **-R***path*
- **/opt/SUNWspro/lib/**
- **/usr/lib** standard UNIX default

The search paths are built into the executable.

4.3.2 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 **-L***dir* 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:

f95 ... **-L***path1* ... **-L***pathn* **...**

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.

4.3.3 Library Search Path and Order—Static Linking

Use the **-l***library* 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% f95 -o pgram part1.o part2.o -L/home/proj/libs -lmylib
```

4.3.3.1 Command-Line Order for -l*library* 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 **-l***library* options as follows:

- Place the **-l***library* option after any **.f**, **.for**, **.F**, **.f95**, or **.o** files.
- If you call functions in **lib***x*, and they reference functions in **lib***y*, then place **-l***x* before **-l***y*.

4.3.3.2 Command-Line Order for -L*dir* Options

The **-L***dir* 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**–l***library* options to which it applies.

4.3.4 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.

4.3.4.1 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**:

```
demo% f95 program.f -R/home/proj/libs -L/home/proj/libs -lmylib
demo% dump -Lv a.out | grep RPATH
[5]      RPATH    /home/proj/libs:/opt/SUNWspro/lib
```

4.3.4.2 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.

4.3.4.3 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:

```
demo% ldd a.out
libfui.so.1 =>   /opt/SUNWspro/lib/libfui.so.1
    libfai.so.1 =>   /opt/SUNWspro/lib/libfai.so.1
    libfai2.so.1 =>  /opt/SUNWspro/lib/libfai2.so.1
    libfsumai.so.1 =>    /opt/SUNWspro/lib/libfsumai.so.1
    libfprodai.so.1 =>   /opt/SUNWspro/lib/libfprodai.so.1
    libfminlai.so.1 =>   /opt/SUNWspro/lib/libfminlai.so.1
    libfmaxlai.so.1 =>   /opt/SUNWspro/lib/libfmaxlai.so.1
```

```
libfminvai.so.1 =>   /opt/SUNWspro/lib/libfminvai.so.1
libfmaxvai.so.1 =>   /opt/SUNWspro/lib/libfmaxvai.so.1
libfsu.so.1 =>   /opt/SUNWspro/lib/libfsu.so.1
libsunmath.so.1 =>   /opt/SUNWspro/lib/libsunmath.so.1
libm.so.1 =>     /usr/lib/libm.so.1
libc.so.1 =>     /usr/lib/libc.so.1
libdl.so.1 =>    /usr/lib/libdl.so.1
/usr/platform/SUNW,Ultra-5_10/lib/libc_psr.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.

4.4 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.

4.4.1 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`**:

```
demo% f95 main.f -lmylibrary crunch.f -o myprog
```

(Incorrect)

```
demo% f95 main.f crunch.f -lmylibrary -o myprog
```

(Correct)

4.4.2 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:

```
demo% cd test_lib
demo% ls
total 14          2 dropx.f      2 evalx.f      2 markx.f
   2 delte.f      2 etc.f        2 linkz.f      2 point.f
```

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:

```
demo% f95 -c *.f
demo% ls
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
demo%
```

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

```
demo% ar cr testlib.a *.o
```

To use this library, either include the library file on the compilation command or use the **-l** and **-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% f95 -o trylib trylib.f test_lib/testlib.a
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 **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, **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 **-l***library* and **-L***path* options. Here, the library's name would have to be changed to conform to the **lib***name***.a** convention:

```
demo% mv test_lib/testlib.a test_lib/libtestlib.a
demo% f95 -o trylib trylib.f -Ltest_lib -ltestlib
```

The **-l***library* and **-L***path* options are used with libraries installed in a commonly accessible directory on the system, like **/usr/local/lib**, so that other users can reference it. For example, if you left **libtestlib.a** in **/usr/local/lib**, other users could be informed to compile with the following command:

```
demo% f95 -o myprog myprog.f -L/usr/local/lib -ltestlib
```

4.4.2.1 Replacement in a Static Library

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

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

```
demo% f95 -c point.f
demo% ar -r testlib.a point.o
```

4.4.2.2 Ordering Routines in a Static Library

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

```
demo% ar -cr mylib.a 'lorder exg.o fofx.o diffz.o | tsort'
```

4.5 Creating Dynamic Libraries

Dynamic library files are built by the linker **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 *shared* library.

A dynamic library offers the following features:

- The object modules are *not* bound into the executable file by the linker during the compile-link sequence; such binding is deferred until runtime.
- A shared library module is bound 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.

4.5.1 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 library 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.

4.5.2 Position-Independent Code and –xcode

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 the **-xcode** compiler option.

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.

Use the compiler flag **-xcode=***v* for specifying the code address space of a binary object. With this 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 the legacy **-pic** flag, and **-xcode=pic32** is equivalent to **-PIC**.)

The **-xcode=pic32** compiler option is similar to **-xcode=pic13**, but allows the global offset table to span the range of 32–bit addresses. See the **f95**(1) man page or the *Fortran User's Guide*, for details.

4.5.3 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.

4.5.3.1 –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:

```
f95 prog.f -Bdynamic -lwells -Bstatic -lsurface
```

4.5.3.2 –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.

4.5.3.3 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:

```
f95 -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:

```
f95 -m64 -o cdf -nolib cdf.o -Bstatic -lsunmath \ -Bdynamic -lm -lc
```

4.5.4 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 **-h***name* option records *name* as the name of the dynamic library being built.

4.5.5 A Simple Dynamic Library

Building a dynamic library requires a compilation of the source files with the **-xcode** option and linker options **-G**, **-ztext**, and **-h***name*. 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% f95 -o libtestlib.so.1 -G -xcode=pic13 -ztext  \
—hlibtestlib.so.1 *.f
```

—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% f95 -o trylib -R"pwd" trylib.f libtestlib.so.1
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
     libfui.so.1 => /opt/SUNWspro/lib/libfui.so.1
     libfai.so.1 => /opt/SUNWspro/lib/libfai.so.1
     libc.so.1 => /usr/lib/libc.so.1
```

Note that the example uses the **-R** option to bind into the executable the path (the current directory) to the dynamic library.

The **file** command shows that the executable is dynamically linked.

4.5.6 Initializing Common Blocks

When building dynamic libraries, insure proper initialization of common blocks (by **DATA** or **BLOCK DATA**) by gathering the initialized common blocks into the same library, and referencing that library before all others.

For example:

```
demo% f95 -G -xcode=pic32 -o init.so blkdat1.f blkdat2.f blkdat3.f
demo% f95 -o prog main.f init.so otherlib1.so otherlib2.so
```

The first compilation creates a dynamic library from files that define common blocks and initialize them in **BLOCK DATA** units. The second compilation creates the executable binary, linking the compiled main program with the dynamic libraries required by the application. Note that the dynamic library that initializes all the common blocks appears first before all the other libraries. This insures the blocks are properly initialized.

4.6 Libraries Provided With Sun Fortran Compilers

The table shows the libraries installed with the compilers.

TABLE 4–1 Major Libraries Provided With the Compilers

Library	Name	Options Needed
f95 support intrinsics	**libfsu**	None
f95 interface	**libfui**	None
f95 array intrinsics libraries	**libf*ai**	None
f95 interval arithmetic intrinsic library	**libifai**	**-xinterval**
Library of Sun math functions	**libsunmath**	None

4.7 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 Sun Studio SDN portal:

`http://developers.sun.com/sunstudio/documentation/ss12/`

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.

◆ ◆ ◆ CHAPTER 5

Program Analysis and Debugging

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

5.1 Global Program Checking (`-Xlist`)

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

`–Xlist` reports errors in alignment, agreement in number and type for subprogram arguments, common block, parameter, and various other kinds of errors.

`–Xlist` also can be used to make detailed source code listings and cross-reference tables.

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

5.1.1 GPC Overview

Global program checking (GPC), invoked by the **`-Xlist`***x* 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

5.1.2 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:

```
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

5.1.2.1 Screen Output

Normally, output listings produced by **-Xlist***x* 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% f95 -Xlisto /dev/tty  any1.f
```

5.1.2.2 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

5.1.2.3 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.

5.1.3 Some Examples of -Xlist and Global Program Checking

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

```
demo% cat Repeat.f
       PROGRAM repeat
         pn1 = 27.005
         CALL subr1 ( pn1 )
         CALL newf ( pn1 )
         PRINT *, pn1
       END

       SUBROUTINE subr1 ( x )
         IF ( x .GT. 1.0 ) THEN
          CALL subr2 ( x * 0.5 )
         END IF
       END

       SUBROUTINE newf( ix )
         INTEGER PRNOK
         IF (ix .eq. 0) THEN
                ix = -1
         ENDIF
         PRINT *, prnok ( ix )
       END

       INTEGER FUNCTION prnok ( x )
         prnok = INT ( x ) + .05
       END
```

```
      SUBROUTINE unreach_sub()
        CALL sleep(1)
      END

      SUBROUTINE subr2 (x)
        CALL subr1(x+x)
      END
```

Example: Use **-XlistX** to show errors, warnings, and cross-reference

```
demo% f95 -XlistX Repeat.f
demo% cat Repeat.lst
Repeat.f                    Mon Mar 18 18:08:27 2002    page 1

FILE  "Repeat.f"
program  repeat
     4           CALL newf ( pn1 )
                           ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer
                 See: "Repeat.f" line #14
     5           PRINT *, pn1
                          ^
**** ERR  #570:  variable "pn1" referenced as real but set as integer in
                 line #4
subroutine  newf
    19           PRINT *, prnok  ( ix )
                                   ^
**** ERR  #418:  argument "ix" is integer, but dummy argument is real
                 See: "Repeat.f" line #22
function  prnok
    23           prnok = INT ( x ) + .05
                 ^
**** WAR  #1024:  suspicious assignment a value of type "real*4" to a
                 variable of type "integer*4"
subroutine  unreach_sub
    26         SUBROUTINE unreach_sub()
                          ^
**** WAR  #338:  subroutine "unreach_sub" never called from program
subroutine  subr2
    31           CALL subr1(x+x)
                 ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #10
                 "Repeat.f" line #3

Cross Reference                Mon Mar 18 18:08:27 2002    page 2
```

```
        C R O S S   R E F E R E N C E   T A B L E

  Source file:   Repeat.f


Legend:
D        Definition/Declaration
U        Simple use
M        Modified occurrence
A        Actual argument
C        Subroutine/Function call
I        Initialization: DATA or extended declaration
E        Occurrence in EQUIVALENCE
N        Occurrence in NAMELIST
L        Use Module



Cross Reference                Mon Mar 18 15:40:57 2002     page 3

P R O G R A M   F O R M

 Program
 -------

repeat          <repeat>        D      1:D

Cross Reference                Mon Mar 18 15:40:57 2002    page 4


 Functions and Subroutines
 -------------------------

INT      intrinsic
                <prnok>         C     23:C

newf             <repeat>        C      4:C
                 <newf>          D     14:D

prnok    int*4   <newf>         DC     15:D     19:C
                 <prnok>        DM     22:D     23:M

sleep            <unreach_sub>            C     27:C

subr1            <repeat>        C      3:C
                 <subr1>         D      8:D
                 <subr2>         C     31:C
```

```
subr2           <subr1>         C     10:C
                <subr2>         D     30:D

unreach_sub     <unreach_sub>            D     26:D

Cross Reference                 Mon Mar 18 15:40:57 2002    page 5


 Variables and Arrays
 --------------------

ix      int*4  dummy
                <newf>       DUMA     14:D     16:U     17:M     19:A

pn1      real*4 <repeat>      UMA      2:M      3:A      4:A      5:U

x       real*4 dummy
                <subr1>        DU      8:D      9:U     10:U
                <subr2>        DU     30:D     31:U     31:U
                <prnok>        DA     22:D     23:A

------------------------------------------------------------------
STATISTIC                    Mon Mar 18 15:40:57 2002    page 6


Date:     Mon Mar 18 15:40:57 2002
Options:  -XlistX
Files:         2 (Sources: 1; libraries: 1)
Lines:        33 (Sources: 33; Library subprograms:1)
Routines:      6 (MAIN: 1; Subroutines: 4; Functions: 1)
Messages:      6 (Errors: 3; Warnings: 3)
```

5.1.4 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.

5.1.4.1 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**.

5.1.4.2 `-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% f95  -Xlistc  -Xlist  any.f

demo% f95 - Xlistc  any.f
```

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

TABLE 5–1 **Basic Xlist** Suboptions

Generated Report	Option
Errors, listing, cross-reference	**–Xlist**
Errors only	**–XlistE**
Errors and source listing only	**–XlistL**
Errors and cross-reference table only	**–XlistX**
Errors and call graph only	**–Xlistc**

The following table shows all **-Xlist** suboptions.

TABLE 5–2 Complete List of **-Xlist** Suboptions

Option	Action
–Xlist (*no suboption)*	Shows errors, listing, and cross-reference table
–Xlistc	Shows call graphs and 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	Shows errors Used alone, **-XlistE** shows only cross-routine errors and does not show a listing or a cross-reference.

TABLE 5–2 Complete List of **-Xlist** Suboptions *(Continued)*

Option	Action
–Xlisterr[*nnn***]**	Suppresses error *nnn* in the verification report Use **-Xlisterr** to suppress a numbered error message from the listing or cross-reference. For example: **-Xlisterr338** suppresses error message 338. To suppress additional specific errors, use this option repeatedly. If *nnn* is not specified, all error messages are suppressed.
–Xlistf	Produces output faster Use **-Xlistf** to produce source file listings and a cross-checking report and to check sources without full compilation.
–Xlisth	Shows errors from cross-checking stop compilation 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	Lists and cross-checks **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	Shows the listing and 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
–Xlistl*n*	Sets page breaks Use **-Xlistl** to set the page length to something other than the default page size. For example, **-Xlistl45** sets the page length to 45 lines. The default is 66. With *n*=0 (**-Xlistl0)** this option shows listings and cross-references with no page breaks for easier on-screen viewing.

TABLE 5–2 Complete List of **-Xlist** Suboptions *(Continued)*

Option	Action
-XlistMP	(**SPARC**) Check consistency of OpenMP directives Use **-XlistMP** to report on any inconsistencies in the OpenMP directives specified in the source code file. See also the *OpenMP API User's Guide* for details.
–Xlisto *name*	Specify the **-Xlist** output report file Use **-Xlisto** to specify the generated report output file. (A space between **o** and *name* is required.) With **-Xlisto name**, the output is to **name** and not to *file***.lst**. To display directly to the screen, use the option: **-Xlisto /dev/tty**
–Xlists	Suppresses unreferenced symbols from cross-reference 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.
–Xlistv*n*	Sets checking "strictness" level *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.

TABLE 5–2 Complete List of **-Xlist** Suboptions *(Continued)*

Option	Action
–Xlistw[*nnn***]**	Sets the width of output lines 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]*	Suppresses 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	Shows just the cross-reference table and errors **-XlistX** produces a cross reference table and cross routine error list but no source listing.

5.2 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.

5.2.1 Subscript Bounds (–C)

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

Example: Index out of range:

```
demo% cat range.f
      REAL a(10,10)
      k = 11
      a(k,2) = 1.0
      END
demo% f95 -o range range.f
demo% range

 ******  FORTRAN RUN-TIME SYSTEM  ******
Subscript out of range. Location: line 3 column 9 of 'range.f'
Subscript number 1 has value 11 in array 'A'
```

```
Abort
demo%
```

5.2.2 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.

5.2.3 Compiler Version Checking (–V)

The **–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.

```
demo% f95 -V wh.f
f95: Sun Fortran 95 7.0 DEV 2002/01/30
f90comp: Sun Fortran 95 7.0 DEV 2002/01/30
f90comp: 9 SOURCE LINES
f90comp: 0 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
ld: Solaris Link Editors: 5.8-1.272
```

5.3 Debugging With dbx

Sun Studio provides a tightly integrated development environment for debugging applications written in Fortran, C, and C++.

The **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
- 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.

For details, see the Sun Studio *Debugging a Program With* **dbx** manual, and the **dbx**(1) man pages.

◆ ◆ ◆ CHAPTER 6

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 *Numerical Computation Guide*.

6.1 Introduction

The Fortran 95 floating-point environment on SPARC processors 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 Fortran 95.

6.2 IEEE Floating-Point Arithmetic

IEEE arithmetic is a relatively new way of dealing with arithmetic operations that result in such problems as invalid operand, division by zero, overflow, underflow, or inexact result. 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:

`big*big = +Inf` *Positive infinity*

`big*(-big) = -Inf` *Negative infinity*

`num/0.0 = +Inf` *Where* num > 0.0

`num/0.0 = -Inf` *Where* num < 0.0

`0.0/0.0 = NaN` *Not a Number*

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 *Numerical Computation Guide*.

6.2.1 –`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 the *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`**. (This is the **`f95`** default.)

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

6.2.2 Floating-Point Exceptions

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

6.2.2.1 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. 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)
```

A Fortran 95 program would need to call **ieee_retrospective** explicitly and compile with **-xlang=f77** to link with the **f77** compatibility library.

Compiling with the **-f77** compatibility flag will enable the Fortran 77 convention of automatically calling **ieee_retrospective** at program termination.

You can turn off any or all of these messages with **ieee_flags()** by clearing exception status flags before the call to **ieee_retrospective**.

6.2.3 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.

6.2.4 Trapping a Floating-Point Exception

f95 differs significantly from the earlier **f77** compiler in the way it handles floating-point exceptions.

The default with **f95** is to automatically trap on division by zero, overflow, and invalid operation. With **f77**, the default was *not* to automatically generate a signal to interrupt the running program for a floating-point exception. The assumption was that trapping would degrade performance while most exceptions were insignificant as long as expected values are returned.

The **f95** command-line option **-ftrap** can be used to change the default. The default for **f95** is **-ftrap=common.** To follow the earlier **f77** default, compile the main program with **-ftrap=%none**.

6.2.5 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), 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.

6.3 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.

    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 **-xtarget=** option permits specification of the runtime hardware. For example, **-xtarget=ultra** would inform the compiler to generate object code that will perform best on an UltraSPARC processor.

    The utility **fpversion** displays which floating-point hardware is installed and indicates the appropriate **-xtarget** value to specify. This utility runs on all Sun SPARC architectures. See **fpversion**(1), the *Fortran User's Guide*, and the *Numerical Computation Guide* for details.

6.3.1 Flags and `ieee_flags()`

The **ieee_flags** function is used to query and clear exception status flags. It is part of the **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 **ieee_flags** is:

```
flags = ieee_flags( action, mode, in, out )
```

Each of the four arguments is a string. The input is *action*, *mode*, and *in*. The output is *out* and *flags*. **ieee_flags** is an integer-valued function. Useful information is returned in *flags* as a set of 1-bit flags. Refer to the man page for **ieee_flags**(3m) for complete details.

Possible parameter values are shown in the following table

TABLE 6–1 **ieee_flags**(*action, mode, in, out*) Argument Values

Argument	Values Allowed
action	**get**, **set**, **clear**, **clearall**

TABLE 6–1 **ieee_flags**(*action, mode, in, out*) Argument Values *(Continued)*

Argument	Values Allowed
mode	**direction**, **exception**
in, out	**nearest**, **tozero**, **negative**, **positive**, **extended**, **double single**, **inexact**, **division**, **underflow**, **overflow**, **invalid all**, **common**

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 **ieee_flags** *in, out* Argument Meanings

Value of *in* and *out*	Refers to
nearest, **tozero**, **negative**, **positive**	Rounding direction
extended, **double**, **single**	Rounding precision
inexact, division, underflow, overflow, invalid	Exceptions
all	All five exceptions
common	Common exceptions: invalid, division, overflow

For example, to determine what is the highest priority exception that has a flag raised, pass the input argument *in* as the null string:

```
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.

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

Example: Clear the **invalid** exception:

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

Example: Clear all exceptions:

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

Example: Set rounding direction to zero:

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

Example: Set rounding precision to **double**:

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

6.3.1.1 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()**:

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

6.3.1.2 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**.

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:

```
#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 *, ' 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**):

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

6.3.2 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:

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

The values available are listed in the following table:

TABLE 6–3 Functions Returning IEEE Values

IEEE Value	Double Precision	Single Precision
infinity	**d_infinity()**	**r_infinity()**
quiet NaN	**d_quiet_nan()**	**r_quiet_nan()**
signaling NaN	**d_signaling_nan()**	**r_signaling_nan()**
min normal	**d_min_normal()**	**r_min_normal()**
min subnormal	**d_min_subnormal()**	**r_min_subnormal()**
max subnormal	**d_max_subnormal()**	**r_max_subnormal()**
max normal	**d_max_normal()**	**r_max_normal()**

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** header file and **floatingpoint**(3f)

6.3.3 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 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()**.

6.3.3.1 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 6–4 Arguments for **ieee_handler**(*action* **,** *exception* **,** *handler*)

Argument	Type	Possible Values
action	**character**	**get**, **set**, or **clear**
exception	**character**	**invalid**, **division**, **overflow**, **underflow**, or **inexact**
handler	Function name	The name of the user handler function or **SIGFPE_DEFAULT**, **SIGFPE_IGNORE**, or **SIGFPE_ABORT**
Return value	**integer**	0 =*OK*

A Fortran 95 routine compiled with **f95** that calls **ieee_handler()** should also declare:

```
#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:

SIGFPE_DEFAULT *or* **SIGFPE_IGNORE**	No action taken when the specified exception occurs.
SIGFPE_ABORT	Program aborts, possibly with dump file, on exception.

6.3.3.2 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:

*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 example would have to be modified to run on 64 bit SPARC architectures 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:

```
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% f95 DetExcHan.f
demo% a.out
Abort
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:

```
demo% cat LocExcHan.F
#include "floatingpoint.h"
      EXTERNAL Exhandler
      INTEGER Exhandler, i, ieee_handler
      REAL:: r = 14.2 ,  s = 0.0 , t
C  Detect division by zero
      i = ieee_handler( 'set', 'division', Exhandler )
      t = r/s
      END

      INTEGER FUNCTION Exhandler( sig, sip, uap)
      INTEGER sig
      STRUCTURE /fault/
                  INTEGER address
      END STRUCTURE
      STRUCTURE /siginfo/
                  INTEGER si_signo
```

```
                INTEGER si_code
                INTEGER si_errno
                RECORD /fault/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      WRITE (*,10)  sip.si_signo, sip.si_code, sip.fault.address
10        FORMAT('Signal ',i4,' code ',i4,' at hex address ', Z8 )
      Exhandler=1
      CALL abort()
      END
demo%  f95 -g LocExcHan.F
demo%  a.out
Signal   8 code    3 at hex address    11230
Abort
demo%
```

In 64–bit SPARC environments, replace the **INTEGER** declarations within each **STRUCTURE** with **INTEGER*8**, and the **i4** formats with **i8**. (Note that this example relies on extensions to the **f95** compiler to accept VAX Fortran **STRUCTURE** statements.)

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()
(dbx) quit
demo%
```

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.

6.4 Debugging IEEE Exceptions

Locating *where* the exception occurred requires exception trapping be enabled. This can be done by either compiling with the **-ftrap=common** option (the default when compiling with **f95**) or by establishing an exception handler routine with **ieee_handler()**. With exception trapping enabled, run the program from **dbx**, using the **dbx catch FPE** command to see where the error occurs.

The advantage of compiling 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: Compiling for and using **dbx**:

```
demo% f95 -g myprogram.f
demo% dbx a.out
Reading symbolic information for a.out
...
(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)
```

If you find that the program terminates with overflow and other exceptions, you can locate the first overflow specifically by calling **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:

```
demo% cat myprog.F
#include “floatingpoint.h”
        program myprogram
...
      ier = ieee_handler(”set’,’overflow’,SIGFPE_ABORT)
...
demo% f95 -g myprog.F
demo% dbx a.out
Reading symbolic information for a.out
...
(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
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.

6.5 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:

```
a = 1.0E-30
b = 1.0E-15
x = a * b
```

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 *Numerical Computation Guide*.

6.5.1 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:

```
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:

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

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.

6.5.2 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:

- *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 **`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.

6.5.3 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:

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

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

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

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.

6.6 Interval Arithmetic

Note: Interval arithmetic is only available on SPARC platforms, currently.

The 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

The **f95** command-line option **-xinterval** enables the interval arithmetic features of the compiler. See the *Fortran User's Guide*.

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

◆ ◆ ◆ CHAPTER 7

7 Porting

This chapter discusses the some issues that may arise when porting "dusty deck" Fortran programs from other platforms to Fortran 95.

Fortran 95 extensions and Fortran 77 compatibility features are described in the *Fortran User's Guide*.

7.1 Carriage-Control

Fortran carriage-control grew out of the limited capabilities of the equipment used when Fortran was originally developed. For similar historical reasons, operating systems derived from the UNIX do not have Fortran carriage control, but you can simulate it with the Fortran 95 compiler in two ways.

- Use the **asa** filter to transform Fortran carriage-control conventions into the UNIX carriage-control format (see the **asa** (1) man page) before printing files with the **lpr** command.
- The FORTRAN 77 compiler **f77** allowed **OPEN(N, FORM='PRINT')** to enable single or double spacing, formfeed, and stripping of column one. This is still available by compiling programs using **FORM='PRINT'** with the **f95 -f77** compatibility flag. The compiler allows you to reopen unit 6 to change the form parameter to **PRINT**, when compiling with **-f77**. For example:

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

You can use **lp**(1) to print a file that is opened in this manner.

7.2 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**(1)):

```
demo% cat redir.data              The data file
 9 9.9

demo% cat redir.f                 The source file
      read(*,*) i, z              The program reads standard input
      print *, i, z
      stop
      end

demo% f95 -o redir redir.f        The compilation step
demo% redir < redir.data                Run with redirection reads data file
  9 9.90000
demo%
```

7.3 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 platform, for example:

```
-fast -m64 -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.

See the *Fortran User's Guide* or the **f95**(1) man page for details.

7.4 Data Representation

The *Fortran User's Guide,* and the *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 95 standard, except when these default sizes are changed by the **-xtypemap** option.
- Character variables can be freely mixed and equivalenced to variables of other types, but be careful of potential alignment problems.
- **f95** IEEE floating-point arithmetic will raise exceptions on overflow or divide by zero and signal **SIGFPE** or trap by default (**-ftrap=common** is the default with **f95**). It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. This is explained in Chapter Chapter 6.
- 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.

7.5 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 (*n***H**) 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 **-xtypemap** command-line flag.)

TABLE 7–1 Maximum Characters in Data Types

	Maximum Number of Standard ASCII Characters			
Data Type	Default	INTEGER:64	REAL:64	DOUBLE:128
BYTE	1	1	1	1
COMPLEX	8	8	16	16

TABLE 7–1 Maximum Characters in Data Types *(Continued)*

	Maximum Number of Standard ASCII Characters			
Data Type	Default	INTEGER:64	REAL:64	DOUBLE:128
COMPLEX*16	16	16	16	16
COMPLEX*32	32	32	32	32
DOUBLE COMPLEX	16	16	32	32
DOUBLE PRECISION	8	8	16	16
INTEGER	4	8	4	8
INTEGER*2	2	2	2	2
INTEGER*4	4	4	4	4
INTEGER*8	8	8	8	8
LOGICAL	4	8	4	8
LOGICAL*1	1	1	1	1
LOGICAL*2	2	2	2	2
LOGICAL*4	4	4	4	4
LOGICAL*8	8	8	8	8
REAL	4	4	8	8
REAL*4	4	4	4	4
REAL*8	8	8	8	8
REAL*16	16	16	16	16

Example: Initialize variables with Hollerith:

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

demo% f95 -o FourA8 FourA8.f
demo% FourA8
abcdefghijklmnopqrstuvwxyz012345!
demo%
```

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

If you pass Hollerith constants as arguments, or if you use them in expressions or comparisons, they are interpreted as character-type expressions. Use the compiler option **`-xhasc=no`** to have the compiler treat Hollerith constants as typeless data in arguments on subprogram calls. This may be needed when porting older Fortran programs.

7.6 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.

7.6.1 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.

7.6.2 Aliasing and the `-xalias` Option

Aliasing occurs when the same storage address is referenced by more than one name. This typically happens with pointers, or 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)
...
```

Many "dusty deck" Fortran programs utilized this sort of aliasing as a way of providing some kind of dynamic memory management that was not available in the language at that time.

Avoid aliasing in all portable code. The results could be unpredictable on some platforms and when compiled with optimization levels higher than **-O2**.

The **f95** compiler assumes it is compiling a standard-conforming program. Programs that do not conform strictly to the Fortran standard can introduce ambiguous situations that interfere with the compiler's analysis and optimization strategies. Some situations can produce erroneous results.

For example, overindexing arrays, use of pointers, or passing global variables as subprogram arguments when also used directly, can result in ambiguous situations that limit the compiler's ability to generate optimal code that will be correct in all situations.

If you know that your program does contain some apparent aliasing situations you can use the **-xalias** option to specify the degree to which the compiler should be concerned. In some cases the program will not execute properly when compiled at optimization levels higher than **-O2** unless the appropriate **-xalias** option is specified.

The option flag takes a comma-separated list of keywords that indicate a type of aliasing situation. Each keyword can be prefixed by **no%** to indicate an aliasing that is not present.

TABLE 7–2 **-xalias** Keywords and What They Mean

-xalias= *keyword*	**Aliasing situation**
dummy	Dummy subprogram arguments can alias each other and global variables.
no%dummy	The Fortran standard is followed and dummy arguments do not alias each other or global variables in the actual call. (This is the default.)
craypointer	The program uses Cray pointers that can point anywhere. (This is the default.)
no%craypointer	Cray pointers always point at distinct memory areas, or are not used.
ftnpointer	Any Fortran 95 pointer can point to any target variable, regardless of type, kind, or rank.
no%ftnpointer	Fortran 95 pointers follow the rules of the standard. (This is the default.)

TABLE 7–2 **-xalias** Keywords and What They Mean *(Continued)*

-xalias= *keyword*	**Aliasing situation**
overindex	There are four overindexing situations that can be caused by violating the subscript bounds in an array reference, and any one or more of these may appear in the program: ■ A reference to an element of an array in a COMMON block could refer to any element in a COMMON block or equivalence group. ■ Passing an element of a COMMON block or equivalence group as an actual argument to a subprogram gives access to any element of that COMMON block or equivalence group. ■ Variables of a sequence derived type are treated as if they were COMMON blocks, and elements of a such a variable may alias other elements of that variable. ■ Individual array subscript bounds may be violated, even though the array reference stays within the array. **overindex** does not apply to array syntax, **WHERE**, and **FORALL** statements. If overindexing occurs in these constructs, they should be rewritten as **DO** loops.
no%overindex	Array bounds are not violated. Array references do not reference other variables. (This is the default.)
actual	The compiler treats actual subprogram arguments as if they were global variables. Passing an argument to a subprogram may result in aliasing through Cray pointers.
no%actual	Passing an argument to a subprogram does not cause further aliasing. (This is the default.)

Here some examples of typical aliasing situations. At the higher optimization levels (**-O3** and above) the **f95** compiler can generate better code if your program does not contain the aliasing syndromes shown below and you compile with **-xalias=no%***keyword*.

In some cases you will need to compile with **-xalias=***keyword* to insure that the code generate will produce the correct results.

7.6.2.1 Aliasing Through Dummy Arguments and Global Variables

The following example needs to be compiled with **-xalias=dummy**

```
parameter (n=100)
integer a(n)
common /qq/z(n)
call sub(a,a,z,n)
...
subroutine sub(a,b,c,n)
```

```
integer a(n), b(n)
common /qq/z(n)
a(2:n) = b(1:n-1)
c(2:n) = z(1:n-1)
```
The compiler must assume that the dummy variables and the common variable may overlap.

7.6.2.2 Aliasing Introduced With Cray Pointers

This example works only when compiled with **`-xalias=craypointer`**, which is the default:

```
parameter (n=20)
integer a(n)
integer v1(*), v2(*)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a)
p2 = loc(a)
a = (/ (i,i=1,n) /)
...
v1(2:n) = v2(1:n-1)
```
The compiler must assume that these locations can overlap.

Here is an example of Cray pointers that do not overlap. In this case, compile with **`-xalias=no%craypointer`** for possibly better performance:

```
parameter (n=10)
integer a(n+n)
integer v1(n), v2(n)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a(1))
p2 = loc(a(n+1))
...
v1(:) = v2(:)
```
The Cray pointers to not point to overlapping memory areas.

7.6.2.3 Aliasing Introduced With Fortran 95 Pointers

Compile the following example with **`-xalias=ftnpointer`**

```
parameter (n=20)
integer, pointer :: a(:)
integer, target :: t(n)
interface
  subroutine sub(a,b,n)
    integer, pointer :: a(:)
    integer, pointer :: b(:)
  end subroutine
```

```
end interface

a => t
a = (/ (i, i=1,n) /)
call sub(a,a,n)
....
end
subroutine sub(a,b,n)
 integer, pointer :: a(:)
 real, pointer :: b(:)
 integer i, mold

 forall (i=2:n)
   a(i) = transfer(b(i-1), mold)
```

The compiler must assume that a and b can overlap.

Note that in this example the compiler must assume that a and b may overlap, even though they point to data of different data types. This is illegal in standard Fortran. The compiler gives a warning if it can detect this situation.

7.6.2.4 Aliasing By Overindexing

Compile the following example with **-xalias=overindex**

```
integer a,z
common // a(100),z
z = 1
call sub(a)
print*, z
subroutine sub(x)
  integer x(10)
  x(101) = 2
```

The compiler may assume that the call to sub may write to z

```
The program prints 2, and not 1, when compiled with -xalias=overindex
```

Overindexing appears in many legacy Fortran 77 programs and should be avoided. In many cases the result will be unpredictable. To insure correctness, programs should be compiled and tested with the **-C** (runtime array bounds checking) option to flag any array subscripting problems.

In general, the **overindex** flag should only be used with legacy Fortran 77 programs. **-xalias=overindex** does not apply to array syntax expressions, array sections, **WHERE**, and **FORALL** statements.

Fortran 95 programs should always conform to the subscripting rules in the Fortran standard to insure correctness of the generated code. For example, the following example uses ambiguous subscripting in an array syntax expression that will *always* produce an incorrect result due to the overindexing of the array:

This example of array syntax overindexing DOES NOT GIVE CORRECT RESULTS!

```
   parameter (n=10)
   integer a(n),b(n)
   common /qq/a,b
   integer c(n)
   integer m, k
   a = (/ (i,i=1,n) /)
   b = a
   c(1) = 1
   c(2:n) = (/ (i,i=1,n-1) /)

   m = n
   k = n + n
C
C   the reference to a is actually a reference into b
C   so this should really be  b(2:n) = b(1:n-1)
C
   a(m+2:k) = b(1:n-1)

C  or doing it in reverse
   a(k:m+2:-1) = b(n-1:1:-1)
```

Intuitively the user might expect array b to now look like array c, but the result is unpredictable

The **xalias=overindex** flag will not help in this situation since the **overindex** flag does not extend to array syntax expressions. The example compiles, but will not give the correct results. Rewriting this example by replacing the array syntax with the equivalent DO loop will work when compiled with **-xalias=overindex**. But this kind of programming practice should be avoided entirely.

7.6.2.5 Aliasing By Actual Arguments

The compiler looks ahead to see how local variables are used and then makes assumptions about variables that will not change over a subprogram call. In the following example, pointers used in the subprogram defeat the compiler's optimization strategy and the results are unpredictable. To make this work properly you need to compile with the **-xalias=actual** flag:

```
 program foo
      integer i
      call take_loc(i)
      i = 1
      print * , i
      call use_loc()
      print * , i
   end

   subroutine take_loc(i)
```

```
      integer i
      common /loc_comm/ loc_i
      loc_i = loc(i)
   end subroutine take_loc

   subroutine use_loc()
      integer vi1
      pointer (pi,vi)
      common /loc_comm/ loc_i
      pi = loc_i
      vi1 = 3
   end subroutine use_loc
```

take_loc takes the address of **i** and saves it away. **use_loc** uses it. This is a violation of the Fortran standard.

Compiling with the **-xalias=actual** flag informs the compiler that all arguments to subprograms should be considered global within the compilation unit, causing the compiler to be more cautious with its assumptions about variables appearing as actual arguments.

Programming practices like this that violate the Fortran standard should be avoided.

7.6.2.6 `-xalias` Defaults

Specifying **-xalias** without a list assumes that your program does not violate the Fortran aliasing rules. It is equivalent to asserting **no%** for all the aliasing keywords.

The compiler default, when compiling without specifying **-xalias**, is:

```
-xalias=no%dummy,craypointer,no%actual,no%overindex,no%ftnpointer
```

If your program uses Cray pointers but conforms to the Fortran aliasing rules whereby the pointer references cannot result in aliasing, even in ambiguous situations, compiling with **-xalias** may result in generating better optimized code.

7.6.3 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.

7.6.3.1 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(I)*BX(I)/2.
  QX(I) = TX**2
END DO
```

7.6.3.2 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:

```
DO       K = 1, N-5, 6
   DO    J = 1, N
      DO I = 1,N
         A(I,J) = A(I,J) + B(I,K  ) * C(K  ,J)
*                        + B(I,K+1) * C(K+1,J)
*                        + B(I,K+2) * C(K+2,J)
*                        + B(I,K+3) * C(K+3,J)
*                        + B(I,K+4) * C(K+4,J)
*                        + B(I,K+5) * C(K+5,J)
      END DO
   END DO
END DO
DO       KK = K,N
   DO    J =1,N
      DO I =1,N
         A(I,J) = A(I,J) + B(I,KK) * C(KK,J)
      END DO
   END DO
END DO
```

should be rewritten the way it was originally intended:

```
DO       K = 1,N
   DO    J = 1,N
      DO I = 1,N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
```

```
      END DO
    END DO
  END DO
```

7.7 Time and Date Functions

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

The time functions supported in the Fortran library are listed in the following table:

TABLE 7–3 Fortran Time Functions

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

For details, see *Fortran Library Reference Manual* or the individual man pages for these functions. Here is a simple example of the use of these time functions (**TestTim.f**):

```
      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, 100000
            q = atan( q )
30      continue
      timediff = dtime( timearray )
      print*, "atan(q) 100000 times took: ", timediff ," seconds"
      end
```

Running this program produces the following results:

```
demo% TimeTest
       Hello, Time Now Is: Thu Feb  8 15:33:36 2001
 See how long 'sleep 4' takes, in seconds
 Elapsed time for sleep 4 was:  4  seconds
 atan(q) 100000 times took:  0.01  seconds
demo%
```

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 **f95** command line, in which case you also get these VMS versions instead of the standard **f95** versions.

TABLE 7–4 Summary: Nonstandard VMS Fortran System Routines

Name	Definition	Calling Sequence	Argument Type
idate	Date as day, month, year	**call idate(d, m, y)**	**integer**
time	Current time as *hhmmss*	**call time(t)**	**character*8**

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) should be used instead. See the *Fortran Library Reference Manual* for details.

7.8 Troubleshooting

Here are a few suggestions for what to try when programs ported to Fortran 95 do not run as expected.

7.8.1 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. For example, 1.9999999e-30 is very near -9.9992112e-33, even though they differ in sign.

  VAX math is not as accurate 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 Chapter Chapter 6 regarding floating-point issues.
- 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 Chapter Chapter 6, 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 x 10^{29} and still have the same floating-point form. Here is an example of different numbers, with the same representation:

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

The output is:

```
99,999,990 x 10^29 = 0.99999993E+37 = 7CF0BDC1
99,999,996 x 10^29 = 0.99999993E+37 = 7CF0BDC1
```

In this example, the difference is 6 x 10^{29}. The reason for this indistinguishable, wide gap is that in IEEE single-precision 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.

7.8.2 Program Fails Without Warning

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

- Compile with minimal optimization (**–01**). 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. (See "7.6.2 Aliasing and the **-xalias** Option" on page 91)

◆ ◆ ◆ CHAPTER 8

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.

8.1 Sun Studio Performance Analyzer

Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools for performance analysis.

Sun Studio software provides a sophisticated pair of tools for collecting and analyzing program performance data:

- The Collector collects performance data on a statistical basis called profiling. The data can include call stacks, microstate accounting information, thread-synchronization delay data, hardware-counter overflow data, address space data, and summary information for the operating system.
- The Performance Analyzer displays the data recorded by the Collector, so you can examine the information. The Analyzer processes the data and displays various metrics of performance at program, function, caller-callee, source-line, and disassembly-instruction levels. These metrics are classed into three groups: clock-based metrics, synchronization delay metrics, and hardware counter metrics.

  The Performance 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.

These two tools help to answer the following kinds of questions:

- How much of the available resources does the program consume?
- Which functions or load objects are consuming the most resources?
- Which source lines and disassembly instructions consume the most resources?

- How did the program arrive at this point in the execution?
- Which resources are being consumed by a function or load object?

The main window of the Performance Analyzer displays a list of functions for the program with exclusive and inclusive metrics for each function. The list can be filtered by load object, by thread, by light-weight process (LWP) and by time slice. For a selected function, a subsidiary window displays the callers and callees of the function. This window can be used to navigate the call tree—in search of high metric values, for example. Two more windows display source code annotated line-by-line with performance metrics and interleaved with compiler commentary, and disassembly code annotated with metrics for each instruction. Source code and compiler commentary are interleaved with the instructions if available.

The Collector and Analyzer are designed for use by any software developer, even if performance tuning is not the developer's main responsibility. They provide a more flexible, detailed and accurate analysis than the commonly used profiling tools **prof** and **gprof**, and are not subject to an attribution error in **gprof**.

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

- Data collection can be done with the **collect**(1) command.
- The Collector can be run from **dbx** using the **collector** subcommands.
- The command-line utility **er_print**(1) prints out an ASCII version of the various Analyzer displays.
- The command-line utility **er_src**(1) displays source and disassembly code listings annotated with compiler commentary but without performance data.

Details can be found in the Sun Studio *Program Performance Analysis Tools* manual.

8.2 The `time` Command

The simplest way to gather basic data about program performance and resource utilization is to use the **time** (1) command or, in **csh**, the **set time** command.

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

```
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 system wallclock resources memory I/O paging

- *user–*

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
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 memory
- *I/O*– 354 reads, 210 writes
- *paging*– 135 page faults, 0 swapouts

8.2.1 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.

8.3 The `tcov` Profiling Command

The **tcov**(1) command, when used with programs compiled with the **-xprofile=tcov** option, 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.

Enhanced statement level coverage is invoked by the **-xprofile=tcov** compiler option and the **tcov -x** option. The output is a copy of the source files annotated with statement execution counts in the margin.

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.

8.3.1 Enhanced `tcov` Analysis

To use **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:

```
demo% f95 -o onetwo -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
demo%
```

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.

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.

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.

For the details, see the **tcov**(1) man page.

◆ ◆ ◆ CHAPTER 9

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.

9.1 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.

9.1.1 Performance Options

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 9–1 Some Effective Performance Options

Action	Option
Uses a combination of optimization options together	**-fast**
Sets compiler optimization level to *n*	**-O***n* (**-O** = **-O3**)
Specifies general target hardware	**-xtarget=***sys*
Specifies a particular Instruction Set Architecture	**-xarch=***isa*
Optimizes using performance profile data (with **-O5**)	**-xprofile=use**
Unrolls loops by *n*	**-unroll=***n*
Permits simplifications and optimization of floating-point	**-fsimple=1|2**
Performs dependency analysis to optimize loops	**-depend**
Performs interprocedural optimizations	**-xipo**

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.

9.1.1.1 `-fast`

This single option selects a number of performance options.

Note – This option is defined as a particular selection of other options that is subject to change from one release to another, and between compilers. Also, some of the options selected by **`-fast`** might not be available on all platforms. Compile with the **`-dryrun`** flag to see the expansion of **`-fast`**.

`-fast` provides high performance for certain benchmark applications. However, the particular choice of options may or may not be appropriate for your application. Use **`-fast`** as a good starting point for compiling your application for best performance. But additional tuning may still be required. If your program behaves improperly when compiled with **`-fast`**, look closely at the individual options that make up**`- fast`** and invoke only those appropriate to your program that preserve correct behavior.

Note also that a program compiled with **`-fast`** may show good performance and accurate results with some data sets, but not with others. Avoid compiling with**`-fast`** those programs that depend on particular properties of floating-point arithmetic.

Because some of the options selected by **`-fast`** have linking implications, if you compile and link in separate steps be sure to link with**`- fast`** also.

`–fast` selects the following options:

- **`–dalign`**
- **`–depend`**
- **`–fns`**
- **`–fsimple=2`**
- **`-ftrap=common`**
- **`–fround=nearest`** (Solaris only)
- **`–libmil`**
- **`–xtarget=native`**
- **`–O5`**
- **`–xlibmopt`** (Solaris only)
- **`-pad=local`** (SPARC only)
- **`-xvector=lib`** (SPARC only)
- **`-nofstore`** (x86 only)
- **`-xregs=frameptr`** (x86 only)

`-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)*. Note also that the exact expansion of **`-fast`** may change with each compiler release. Compiling with **`-dryrun`** will show the expansion of all command-line flags.

Following **-fast** with additional options adds further optimizations. For example:

f95 -fast -m64 ...

compiles for a 64-bit enabled platform.

Because **-fast** invokes **-dalign**, **-fns**, **-fsimple=2**, programs compiled with**- fast** can result in nonstandard floating-point arithmetic, nonstandard alignment of data, and nonstandard ordering of expression evaluation. These selections might not be appropriate for most programs.

9.1.1.2 -O*n*

The compiler performs no optimizations unless a **-O** option is specified explicitly (or implicitly with macro options like **-fast**). In nearly all cases, specifying an optimization level at 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.)

9.1.1.3 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***n* flag on the compiler command line, but must be used with the **-xmaxopt=***n* flag to set a *maximum* optimization level. See the **f95**(1) man page for details.

9.1.1.4 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.)

9.1.1.5 -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.

9.1.1.6 -depend

Adding **-depend** to optimization levels **-O3** and higher extends the compiler's ability to optimize DO loops and loop nests. With this option, the optimizer analyzes inter-iteration data dependences to determine whether or not certain transformations of the loop structure can be performed. Only loops without data dependences can be restructured. However, the added analysis might increase compilation time.

9.1.1.7 -fsimple=2

Unless directed to, the compiler does not attempt to simplify floating-point computations (the default is **-fsimple=0**). **-fsimple=2** enables the optimizer to make aggressive 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. See the *Fortran User's Guide* for important information about this option.

9.1.1.8 `-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*>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 if it is appropriate to execute the unrolled loop. 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 **`-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.

9.1.1.9 `-xtarget=`*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.

UltraSPARC-III and UltraSPARC-IV Support

Both the **-xtarget** and **-xchip** flags accept **ultra3** and **ultra3** variants and will generate optimized code for UltraSPARC-III and UltraSPARC-IV processors. When compiling and running an application on the latest UltraSPARC platforms, specify the **-fast** flag to automatically select the proper compiler optimization options for that platform.

For cross-compilations (compiling on a platform other than the latest UltraSPARC platforms but generating binaries intended to run on an UltraSPARC-III processor), use these flags:

```
-fast -xtarget=ultra3
```

Use **-m64** to compile for 64-bit code generation.

See the *Fortran User's Guide* for a list of **-xtarget** flags for the latest UltraSPARC processors.

Performance profiling, with **-xprofile=collect:** and **-xprofile=use:**, is particularly effective on the UltraSPARC-III and UltraSPARC-IV platforms because it allows the compiler to identify the most frequently executed sections of the program and perform localized optimizations to best advantage.

64-Bit x86 Platform Support

The Sun Studio Fortran compiler supports the compilation of 32-bit and 64-bit code for Solaris and Linux x86 platforms.

The **-xtarget=pentium3** flag expands to: **-xarch=sse -xchip=pentium3 -xcache=16/32/4:256/32/4**.

For Pentium 4 systems, **-xtarget=pentium4** expands to: **-xarch=sse2 -xchip=pentium4 -xcache=8/64/4:256/128/8.**

A new **-m64** option specifies compilation for the 64-bit x64 instruction set.

A new **-xtarget** option, **-xtarget=opteron**, specifies the **-xarch**, **-xchip**, and **-xcache** settings for 32-bit AMD compilation.

You must specify **-m64** after **-fast** and **-xtarget** on the command line to generate 64-bit code. The **-xtarget** option does not automatically generate 64-bit code. The **-fast** option also results in 32-bit code because it is a macro which also defines an **-xtarget** value. All the current **-xtarget** values (except **-xtarget=native64** and **-xtarget=generic64**) result in 32- bit code, so it is necessary to specify **-xarch=m64** after (to the right of) **-fast** or **-xtarget** to compile 64-bit code, as in:

```
% f95 -fast -m64 or % f95 -xtarget=opteron -m64
```

The compilers now predefine **__amd64** and **__x86_64** when you specify **-xarch=amd64**.

Additional information about compilation and performance on 32-bit and 64-bit x86 platforms can be found in the *Fortran User's Guide*.

9.1.1.10 Interprocedural Optimization With -xipo

This new **f95** compiler flag, introduced with the release of Forte Developer 6 update 2, performs whole-program optimizations by invoking an interprocedural analysis pass. Unlike **-xcrossfile**, **-xipo** optimizes across all object files at the link step and is not limited to just the source files on the compile command.

-xipo is particularly useful when compiling and linking large multi-file applications. Object files compiled with **-xipo** have analysis information saved within them. This enables interprocedural analysis across source and pre-compiled program files.

For details on how to use interprocedural analysis effectively, see the *Fortran User's Guide*.

9.1.1.11 Add PRAGMA ASSUME Assertions

By adding **ASSUME** directives at strategic points in the source code you can help guide the compiler's optimization stragegy by revealing important information about the program that is not determinable any other way. For example, you can let the compiler know that the trip count of a **DO** loop is always greater than a value, or that there is a high probability that an IF branch will not be taken. The compiler can use this information to generate better code, based on these assertions.

As an added bonus, the programmer can use the **ASSUME** pragma to validate the exectution of the program by enabling warning messages to be issued whenever an assertion turns out to be false at run time.

For details, see the description of the **ASSUME** pragma in Chapter 2 of the *Fortran User's Guide*, and the **-xassume_control** compiler command-line option in Chapter 3 of that manual.

9.1.2 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.

9.1.3 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.

9.1.4 Eliminating Performance Inhibitors

Use the Sun Studio 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:

9.1.4.1 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.

9.1.4.2 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.

9.1.4.3 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.

9.1.5 Viewing Compiler Commentary

If you compile with the **-g** debugging option, you can view source code annotations generated by the compiler by using the **er_src**(1) utility, part of the Sun Studio Performance Analysis Tools. This utility can also be used to view the source code annotated with the generated assembly language. Here is an example of the commentary produced by er_src on a simple do loop:

```
demo% f95 -c -g -O4 do.f
demo% er_src do.o
Source file: /home/user21/do.f
Object file: do.o
Load Object: do.o

     1.         program do
     2.         common aa(100),bb(100)

   Function x inlined from source file do.f into the code for the following line
   Loop below pipelined with steady-state cycle count = 3 before unrolling
   Loop below unrolled 5 times
   Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
     3.         call x(aa,bb,100)
     4.         end
     5.                 subroutine x(a,b,n)
     6.                 real a(n), b(n)
     7.                 v = 5.
     8.                 w = 10.
```

```
Loop below pipelined with steady-state cycle count = 3 before unrolling
Loop below unrolled 5 times
Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
  9.                do 1 i=1,n
 10. 1                      a(i) = a(i)+v*b(i)
 11.                return
 12.                end
```

Commentary messages detail the optimization actions taken by the compiler. In the example we can see that the compiler has inlined the call to the subroutine and unrolled the loop 5 times. Reviewing this information might provide clues as to further optimization strategies you can use.

For detailed information about compiler commentary and disassembled code, see the Sun Studio *Performance Analyzer* manual.

9.2 Further Reading

The following reference books provide more details:

- *High Performance Computing,* by Kevin Dowd and Charles Severance, O'Reilly & Associates, 2nd Edition, 1998
- *Techniques for Optimizing Applications: High Performance Computing*, by Rajat Garg and Ilya Sharapov, Sun Microsystems Press Blueprint, 2001

◆ ◆ ◆ CHAPTER 10

10 Parallelization

This chapter presents an overview of multiprocessor parallelization and describes the capabilities of Fortran 95 on Solaris SPARC and x86 multiprocessor platforms.

See also *Techniques for Optimizing Applications: High Performance Computing* by Rajat Garg and Ilya Sharapov, a Sun Microsystems BluePrints publication (`http://www.sun.com/blueprints/pubs.html`)

10.1 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.

`f95` 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.

The Fortran 95 compiler provides explicit parallelization by implementing the OpenMP 2.0 Fortran API directives. For legacy programs, **`f95`** also accepts the older Sun and Cray style directives, but use of these directives is now deprecated. OpenMP has become an informal standard for explicit parallelization in Fortran 95, C, and C++ and is recommended over the older directive styles.

For information on OpenMP, see the *OpenMP API User's Guide,* or the OpenMP web site at `http://www.openmp.org`.

10.1.1 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).

10.1.2 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 **`$PARALLEL`** or **`$OMP_NUM_THREADS`** to 1; see "10.1.5 Number of Threads" on page 124).
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.

10.1.3 Data Dependence 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 dependence issues arise include recurrence, reduction, indirect addressing, and data dependent loop iterations.

10.1.3.1 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.

10.1.3.2 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:

```
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.

10.1.3.3 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:

```
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.

10.1.3.4 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.

10.1.4 Compiling for Parallelization

The Sun Studio compilers support the OpenMP parallelization model natively as the primary parallelization model. For information on OpenMP parallelization, see the *OpenMP API User's Guide*. Sun and Cray-style parallelization refer to legacy applications and are no longer supported by current Sun Studio compilers.

.

TABLE 10–1 Fortran 95 Parallelization Options

Option	Flag
Automatic (*only*)	**`-autopar`**
Automatic and Reduction	**`-autopar -reduction`**
Show which loops are parallelized	**`-loopinfo`**
Show warnings with explicit	**`-vpara`**
Allocate local variables on stack	**`-stackvar`**
Compile for OpenMP parallelization	**`-xopenmp`**

Notes on these options:

- Many of these options have equivalent synonyms, such as **`-autopar`** and **`-xautopar`**. Either may be used.
- The compiler prof/gprof profiling options **`-p, -xpg`**, and **`-pg`** should not be used along with any of the parallelization options. The runtime support for these profiling options is not thread-safe. Invalid results or a segmentation fault could occur at runtime.
- **`-reduction`** requires **`-autopar`**.

- **-autopar** includes **-depend** and loop structure optimization.
- **-noautopar**, **-noreduction** are the negations.
- Parallelization options can be in any order, but they must be all lowercase.
- Reduction operations are not analyzed in explicitly parallelized loops.
- **-xopenmp** also invokes **-stackvar** automatically.
- The options **-loopinfo** and **-vpara** must be used in conjunction with one of the parallelization options.

10.1.5 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 number of available virtual processors on the target platform.

The following example shows how to set it:

```
demo% setenv OMP_NUM_THREADS 4          C shell
```

-or-

```
demo$ OMP_NUM_THREADS=4                 Bourne/Korn shell
demo$ export OMP_NUM_THREADS
```

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:

```
demo% psrinfo
0       on-line   since 03/18/2007 15:51:03
1       on-line   since 03/18/2007 15:51:03
2       on-line   since 03/18/2007 15:51:03
3       on-line   since 03/18/2007 15:51:03
```

10.1.6 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 8 megabytes on 64–bit SPARC and 64-bit x86 platforms, 4 megabytes otherwise. The size is set with the **STACKSIZE** environment variable:

```
demo% setenv STACKSIZE 8192     <- Set thread stack size to 8 Mb    C shell
                          -or-
demo$ STACKSIZE=8192             Bourne/Korn Shell
demo$ export STACKSIZE
```

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.

10.2 Automatic Parallelization

With the **`-autopar`** option, the **`f95`** compiler automatically finds 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.

10.2.1 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.

10.2.2 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:

```
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.

10.2.3 Automatic Parallelization Criteria

DO loops that have no cross-iteration data dependencies are automatically parallelized by **-autopar**. 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.

10.2.3.1 Apparent Dependencies

The compilers may automatically eliminate a reference that appears to create a data dependence 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:

```
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-1
    c(i,j) = a(j+1) + 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 dependence, the compiler generates temporary private copies of the array to make the outer loop iterations independent.

10.2.3.2 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

10.2.3.3 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.

10.2.4 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:

```
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 10–2, the compiler will parallelize it if **-reduction** is specified.

10.2.4.1 Recognized Reduction Operations

The following table lists the reduction operations that are recognized by the compiler.

TABLE 10–2 Recognized Reduction Operations

Mathematical Operations	Fortran Statement Templates
Sum	`s = s + v(i)`
Product	`s = s * v(i)`
Dot product	`s = s + v(i) * u(i)`
Minimum	`s = amin( s, v(i))`
Maximum	`s = amax( s, v(i))`

OR	`do i = 1, n`
	`b = b .or. v(i)`
	`end do`

AND	`b = .true.`
	`do i = 1, n`
	`b = b .and. v(i)`
	`end do`

TABLE 10–2 Recognized Reduction Operations *(Continued)*

Mathematical Operations	Fortran Statement Templates

Count of non-zero elements	`k = 0`
	`do i = 1, n`
	`if(v(i).ne.0) k = k + 1`
	`end do`

All forms of the **MIN** and **MAX** function are recognized.

10.2.4.2 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: Roundoff, get the sum of 100,000 random numbers between– 1 and +1:

```
demo% cat t4.f
      parameter ( n = 100000 )
      double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
      s = d_lcrans ( v, n, lb, ub ) ! Get n random nos. between -1 and +1
      s = 0.0
      do i = 1, n
        s = s + v(i)
      end do
      write(*, '(" s = ", e21.15)') s
      end
demo% f95 -O4 -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.

Number of Processors	Output
1	`s = 0.568582080884714E+02`

Number of Processors	Output
2	`s = 0.568582080884722E+02`
3	`s = 0.568582080884721E+02`
4	`s = 0.568582080884724E+02`

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*.

10.3 Explicit Parallelization

This section describes the source code directives recognized by **`f95`** to explicitly indicate which loops to parallelize and what strategy to use.

The Fortran 95 compiler now fully supports the OpenMP Fortran API as the primary parallelization model. See the *OpenMP API User's Guide* for additional information..

Legacy Sun-style and Cray-style parallelization directives are no longer supported by Sun Studio compilers on SPARC platforms, and are not accepted by the compilers on x86 platforms.

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. Compile with **`-xopenmp`** to enable recognition of OpenMP Fortran 95 directives and generation of parallelized DO loop code. 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.

10.3.1 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.

10.3.1.1 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.

10.3.1.2 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 **PARALLEL DO** or **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 "10.1.6 Stacks, Stack Sizes, and Parallelization" on page 124 about increasing the size of the stack.

10.3.1.3 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** and **-loopinfo**, you will get diagnostic messages if the compiler detects a problem while explicitly parallelizing a loop.

The following table lists typical parallelization problems detected by the compiler:

TABLE 10–3 Explicit Parallelization Problems

Problem	Parallelized	Warning Message
Loop is nested inside another loop that is parallelized.	No	No
Loop is in a subroutine called within the body of a parallelized loop.	No	No
Jumping out of loop is allowed by a flow control statement.	No	Yes
Index variable of loop is subject to side effects.	Yes	No
Some variable in the loop has a loop-carried dependency.	Yes	Yes
I/O statement in the loop—*usually unwise, because the order of the output is not predictable.*	Yes	No

Example: Nested loops:

```
      ...
!$OMP PARALLEL DO
      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
      ...
!$OMP PARALLEL DO
      do 100 i = 1, 200       <-parallelized
        ...
        call calc (a, x)
        ...
100      continue
      ...
subroutine calc ( b, y )
      ...
!$OMP PARALLEL DO
      do 1 m = 1, 1000        <-not parallelized
        ...
1      continue
      return
      end
```

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:

```
!$omp parallel do
      do i = 1, 1000     !   Not parallelized, error issued
        ...
        if (a(i) .gt. min_threshold ) go to 20
        ...
      end do
20      continue
      ...
```

The compiler issues an error diagnostic if there is a jump outside a loop marked for parallelization.

Example: A variable in a loop has a loop-carried dependency:

```
demo% cat vpfn.f
      real function fn (n,x,y,z)
      real y(*),x(*),z(*)
      s = 0.0
```

```
!$omp parallel do private(i,s) shared(x,y,z)
      do  i = 1, n
          x(i) = s
          s = y(i)*z(i)
      enddo
      fn=x(10)
      return
      end
demo% f95 -c -vpara -loopinfo -openmp -O4 vpfn.f
"vpfn.f", line 5: Warning: the loop may have parallelization inhibiting reference
"vpfn.f", line 5: PARALLELIZED, user pragma used
```

Here the loop is parallelized but the possible loop carried dependency is diagnosed in a warning. However, be aware that not all loop dependencies can be diagnosed by the compiler.

10.3.1.4 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

```
!$OMP PARALLEL DO PRIVATE(k)
      do i = 1, 10     !  Parallelized
        k = i
        call show ( k )
      end do
      end
      subroutine show( j )
      write(6,1) j
1      format('Line number ', i3, '.')
      end
demo% f95 -openmp t13.f
demo% setenv PARALLEL 4
demo% a.out

Line number 9.
Line number 4.
Line number 5.
Line number 6.
Line number 1.
Line number 2.
Line number 3.
Line number 7.
Line number 8.
```

However, I/O that is recursive, where an I/O statement contains a call to a function that itself does I/O, will cause a runtime error.

10.3.2 OpenMP Parallelization Directives

OpenMP is a parallel programming model for multi-processor platforms that is becoming standard programming practice for Fortran 95, C, and C++ applications. It is the preferred parallel programming model for Sun Studio compilers.

To enable OpenMP directives, compile with the **`-openmp`** option flag. Fortran 95 OpenMP directives are identified with the comment-like sentinel **`!$OMP`** followed by the directive name and subordinate clauses.

The **`!$OMP PARALLEL`** directive identifies the parallel regions in a program. The **`!$OMP DO`** directive identifies **`DO`** loops within a parallel region that are to be parallelized. These directives can be combined into a single **`!$OMP PARALLEL DO`** directive that must be placed immediately before the **`DO`** loop.

The OpenMP specification includes a number of directives for sharing and synchronizing work in a parallel region of a program, and subordinate clauses for data scoping and control.

One major difference between OpenMP and legacy Sun-style directives is that OpenMP requires explicit data scoping as either *private* or *shared*, but and automatic scoping feature is provided.

For more information, including guidelines for converting legacy programs using Sun and Cray parallelization directives, see the *OpenMP API User's Guide*.

10.4 Environment Variables

There are a number of environment variables used with parallelization: **`OMP_NUM_THREADS, SUNW_MP_WARN, SUNW_MP_THR_IDLE, SUNW_MP_PROCBIND, STACKSIZE`**, and others. They are described in the *OpenMP API User's Guide*.

10.5 Debugging Parallelized Programs

Fortran source code:

```
real x / 1.0 /, y / 0.0 /
print *, x/y
end
character  string*5, out*20
```

```
    double precision value
    external exception_handler
    i = ieee_handler('set', 'all', exception_handler)
    string = '1e310'
    print *, 'Input string ', string, ' becomes: ', value
    print *, 'Value of 1e300 * 1e10 is:', 1e300 * 1e10
    i = ieee_flags('clear', 'exception', 'all', out)
    end

    integer function exception_handler(sig, code, sigcontext)
    integer sig, code, sigcontext(5)
    print *, '*** IEEE exception raised!'
    return
    end
```

Runtime output:

```
*** IEEE exception raised!
 Input string 1e310 becomes:  Infinity
 Value of 1e300 * 1e10 is: Inf
 Note: Following IEEE floating-point traps enabled;
   see ieee_handler(3M):
 Inexact;  Underflow;  Overflow;  Division by Zero;  Invalid
   Operand;
 Sun's implementation of IEEE arithmetic is discussed in
  the Numerical Computation Guide.
Debugging Parallelized Programs
```

Debugging parallelized programs requires some extra effort. The following schemes suggest ways to approach this task.

10.5.1 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 automatic parallelization with **–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 **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 **–autopar**.

  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.

- Run loops *backward* serially.

  Replace **DO I=1,N** with **DO I=N,1,-1**. Different results point to data dependencies.

- Avoid using the loop index.

Replace:

```
DO I=1,N
  ...
  CALL SNUBBER(I)
  ...
ENDDO
```

With:

```
  DO I1=1,N
  I=I1
  ...
  CALL SNUBBER(I)
  ...
ENDDO
```

10.6 Further Reading

The following provide more information:

- *OpenMP API User's Guide*
- *Techniques for Optimizing Applications: High Performance Computing*, by Rajat Garg and Ilya Sharapov, Sun Microsystems Press Blueprint, 2001.
- *High Performance Computing*, by Kevin Dowd and Charles Severance, O'Reilly and Associates, 2nd Edition, 1998.
- *Parallel Programming in OpenMP,* by Rohit Chandra et al., Morgan Kaufmann Publishers, 2001.
- *Parallel Programming*, by Barry Wilkinson, Prentice Hall, 1999.

◆ ◆ ◆ CHAPTER 11

C-Fortran Interface

This chapter treats issues regarding Fortran and C interoperability and applies only to the specifics of the Sun Studio Fortran 95, and C compilers.

"11.9 Fortran 2003 Interoperability With C" on page 165 discusses briefly the C binding features proposed in Section 15 of the Fortran 2003 standard. (The standard is available at the international Fortran standards web site, `http://www.j3-fortran.org`). The Fortran 95 compiler implements these features, as described in the standard.

Except where noted, 32-bit x86 processors are treated the same as 32-bit SPARC processors. The same is true for 64-bit x86 and 64-bit SPARC processors, with the exception that REAL*16 and COMPLEX*32 data types are not defined for x86 systems and are only available on SPARC.

11.1 Compatibility Issues

Most C-Fortran interfaces must agree in all of these aspects:

- Function and subroutine definitions and calls
- Data type compatibility
- Argument passing, either by reference or by value
- Order of arguments
- Procedure name, either uppercase, lowercase, or with a trailing underscore (_)
- Passing the right library references to the linker

Some C-Fortran interfaces must also agree on:

- Array indexing and order
- File descriptors and **`stdio`**
- File permissions

11.1.1 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, **void** functions do not return a value.
- In Fortran, a function passes a return value, but a subroutine generally 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** otherwise. 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 the **SUBROUTINE** statement, a zero is returned.

11.1.2 Data Type Compatibility

Table 11–2 summarizes the data sizes and default alignments for Fortran 95 data types compared with C. It assumes no compilation options affecting alignment or promoting default data sizes are applied. Note the following:

- C data types **int**, **long int**, and **long** are equivalent (4 bytes) in a 32-bit environment. However, in a 64-bit environment, **long** and pointers are 8 bytes. This is referred to as the LP64 data model.
- **REAL*16** and **COMPLEX*32**, in a 64-bit SPARC environment and when compiling with any **-m64** option, are aligned on 16-byte boundaries.
- Alignments marked 4/8 indicate that alignment is 8-bytes by default, but on 4-byte boundaries in COMMON blocks. The maximum default alignment in COMMON is 4-bytes. 4/8/16 indicates alignments on 16-byte boundaries when compiling with **-m64** option.
- **REAL(KIND=16)**, **REAL*16**, **COMPLEX(KIND=16)**, **COMPLEX*32**, are only available on SPARC platforms.
- 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 a Fortran 95 routine to a C routine by using **%VAL(***arg***)** at the call site. You can pass arguments by value from C to Fortran 95 provided the Fortran routine has an explicit interface block that declares the dummy argument with the **VALUE** attribute.
- Components of numeric sequence types are aligned the same way as common blocks, and are also affected by the **-aligncommon** option.A *numeric sequence type* is a sequence type where all the components are of type default integer, default real, double-precision real, default complex, or default logical, and are not pointers.
- Components of data types that are not numeric sequence types are aligned on natural alignments in most cases, except QUAD variables. For quad-precision variables, the alignment is different between 32–bit and 64–bit SPARC platforms.
- Components of VAX structures and data types defined with the BIND(C) attribute always have the same alignment as C structures on all platforms.

TABLE 11–1 Data Sizes and Alignment—(in Bytes) Pass by Reference (**f95** and **cc**)

Fortran 95 Data Type	C Data Type	Size	Alignment
BYTE x	**char x**	1	1
CHARACTER x	**unsigned char x ;**	1	1
CHARACTER (LEN=*n***) x**	**unsigned char x[***n***] ;**	*n*	1
COMPLEX x	**struct {float r,i;} x;**	8	4
COMPLEX (KIND=4) x	**struct {float r,i;} x;**	8	4
COMPLEX (KIND=8) x	**struct {double dr,di;} x;**	16	4/8
COMPLEX (KIND=16) x *(SPARC)*	**struct {long double, dr,di;} x;**	32	4/8/16
DOUBLE COMPLEX x	**struct {double dr, di;} x;**	16	4/8
DOUBLE PRECISION x	**double x ;**	8	4
REAL x	**float x ;**	4	4
REAL (KIND=4) x	**float x ;**	4	4
REAL (KIND=8) x	**double x ;**	8	4/8
REAL (KIND=16) x *(SPARC)*	**long double x ;**	16	4/8/16
INTEGER x	**int x ;**	4	4

TABLE 11–1 Data Sizes and Alignment—(in Bytes) Pass by Reference (**f95** and **cc**) *(Continued)*

Fortran 95 Data Type	C Data Type	Size	Alignment
`INTEGER (KIND=1) x`	`signed char x ;`	1	4
`INTEGER (KIND=2) x`	`short x ;`	2	4
`INTEGER (KIND=4) x`	`int x ;`	4	4
`INTEGER (KIND=8) x`	`long long int x;`	8	4
`LOGICAL x`	`int x ;`	4	4
`LOGICAL (KIND=1) x`	`signed char x ;`	1	4
`LOGICAL (KIND=2) x`	`short x ;`	2	4
`LOGICAL (KIND=4) x`	`int x ;`	4	4
`LOGICAL (KIND=8) x`	`long long int x;`	8	4

11.1.3 Case Sensitivity

C and Fortran take opposite perspectives on case sensitivity:

- C is case sensitive—case matters.
- Fortran ignores case by default.

The **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 -U** compiler option.

11.1.4 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. Almost 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 **BIND(C)** attribute declaration to indicate that an external function is a C language function.
- Use the **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 **BIND(C)** attribute declaration to avoid underscores. **BIND(C)** declares the C external functions that can be called from Fortran, and the Fortran routines that can be called from C as arguments. The Fortran compiler does not append an underscore as it ordinarily does with external names. The **BIND(C)** must appear in each subprogram that contains such a reference. The conventional usage is:

```
FUNCTION ABC
 EXTERNAL XYZ
 BIND(C) ABC, XYZ
```

Here the user has specified not only that **XYZ** is an external C function, but that the Fortran caller, **ABC**, should be callable from a C function. If you use **BIND(C)**, the C function does not need an underscore appended to the function name.

11.1.5 Argument-Passing by Reference or Value

In general, Fortran routines pass arguments by reference. In a call, if you enclose an argument with the nonstandard function **%VAL()**, the calling routine passes it by value.

The standard Fortran 95 way to pass arguments by value is the VALUE attribute and through INTERFACE blocks. See "11.4 Passing Data Arguments by Value" on page 157.

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.

11.1.6 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:

This Fortran code fragment:	Is equivalent to this in C:

`CHARACTER*7 S`
`INTEGER B(3)`
`...`
`CALL SAM( S, B(2) )`
	`char s[7];`
	`int b[3];`
	`...`
	`sam_( s, &b[1], 7L ) ;`

11.1.7 Array Indexing and Order

Array indexing and order differ between Fortran and C.

11.1.7.1 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:

```
INTEGER B(0:2)
```

This way, the Fortran element **B(1)** is equivalent to the C element **b[1]**.

11.1.7.2 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 are stored in row-major order: **A[3][2]**

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

This does not present a problem for one-dimensional arrays. However, with multi-dimensional 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.

11.1.8 File Descriptors and `stdio`

Fortran I/O channels are in terms of unit numbers. The underlying SunOS operating 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 the following table.

TABLE 11–2 Comparing I/O Between Fortran and C

	Fortran Units	Standard I/O File Pointers	File Descriptors
Files Open	Opened for reading and writing	Opened for reading, or for writing, or for both; or opened for appending; See **`open`**(2)	Opened for reading, or for writing, or opened for both
Attributes	Formatted or unformatted	Always unformatted, but can be read or written with format-interpreting routines	Always unformatted
Access	Direct or sequential	Direct access if the physical file representation is direct access, but can always be read sequentially	Direct access if the physical file representation is direct access, but can always be read sequentially
Structure	Record	Byte stream	Byte stream
Form	Arbitrary nonnegative integers from 0-2147483647	Pointers to structures in the user's address space	Integers from 0-1023

11.1.9 Libraries and Linking With the `f95` Command

To link the proper Fortran and C libraries, use the **`f95`** command to invoke the linker.

Example 1: Use the compiler to do the linking:

```
demo% cc -c someCroutine.c
demo% f95 theF95routine.f someCroutine.o    The linking step
demo% a.out
```

```
 4.0 4.5
 8.0 9.0
demo%
```

11.2 Fortran Initialization Routines

Main programs compiled by **f95** call dummy initialization routine **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 you can use to supply your own routines, in C, to initialize a program in any customized manner before the program starts up.

One possible use of these initialization routines to call **setlocale** for an internationalized Fortran program. Because **setlocale** does not work if **libc** is statically linked, only Fortran programs that are dynamically linked with **libc** should be internationalized.

The source code for the **init** routines in the library is

```
void f90_init(int *argc_ptr, char ***argv_ptr, Char ***envp_ptr) {}
```

f90_init is called by **f95** main programs. The arguments are set to the address of **argc**, the address of **argv**, and the address of **envp**.

11.3 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*n function.

11.3.1 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 11–3 Passing Simple Data Types

Fortran calls C	C calls Fortran

`integer i`
`real r`
`external CSim`
`i = 100`
`call CSim(i,r)`
`...`
 	
`---------------------------`
 	
`void csim_(int *i, float *r)`
`{`
`*r = *i;`
`}`
	`int i=100;`
	`float r;`
	`extern void fsim_(int *i, float *r);`
	`fsim_(&i, &r);`
	`...`
	
	`----------------------------`
	
	`subroutine FSim(i,r)`
	`integer i`
	`real r`
	`r = i`
	`return`
	`end`

11.3.2 COMPLEX Data

Pass a Fortran COMPLEX data item as a pointer to a C struct of two float or two double data types:

TABLE 11–4 Passing COMPLEX Data Types

Fortran calls C	C calls Fortran

`complex w`
`double complex z`
`external CCmplx`
`call CCmplx(w,z)`
`...`
 	
`-----------------------------`
 	
`struct cpx {float r, i;};`
`struct dpx {double r,i;};`
`void ccmplx_(`
`struct cpx *w,`
`struct dpx *z)`
`{`
`w -> r = 32.;`
`w -> i = .007;`
`z -> r = 66.67;`
`z -> i = 94.1;`
`}`
	`struct cpx {float r, i;};`
	`struct cpx d1;`
	`struct cpx *w = &d1`
	`struct dpx {double r, i;};`
	`struct dpx d2;`
	`struct dpx *z = &d2`
	`fcmplx_( w, z );`
	`...`
	
	`---------------`
	
	`subroutine FCmplx( w, z )`
	`complex w`
	`double complex z`
	`w = (32., .007)`
	`z = (66.67, 94.1)`
	`return`
	`end`

In 64-bit environments, **COMPLEX** values are returned in registers.

11.3.3 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 11–5 Passing a CHARACTER String

Fortran call:	C equivalent:

`CHARACTER*7 S`
`INTEGER B(3)`
`...`
`CALL CSTRNG( S, B(2) )`
`...`
	`char s[7];`
	`int b[3];`
	`...`
	`cstrng_( s, &b[1], 7L );`
	`...`

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.

The call for a character array looks identical to the call for a single character variable. The starting address of the array is passed, and the length that it uses is the length of a single element in the array.

11.3.4 One-Dimensional Arrays

Array subscripts in C start with 0.

TABLE 11–6 Passing a One-Dimensional Array

Fortran calls C	C calls Fortran

`integer i, Sum`
`integer a(9)`
`external FixVec`
`...`
`call FixVec ( a, Sum )`
`...`
 	
`----------`
 	
`void fixvec_ (`
`int v[9], int *sum )`
`{`
`  int i;`
`  *sum = 0;`
`  for ( i = 0; i <= 8; i++ )`
`  *sum = *sum + v[i];`
`}`
	`extern void vecref_`
	`  ( int[], int * );`
	`...`
	`int i, sum;`
	`int v[9] = ...`
	`vecref_( v, &sum );`
	`...`
	
	`----------`
	
	`subroutine VecRef( v, total)`
	`  integer i, total, v(9)`
	`  total = 0`
	`  do i = 1,9`
	`  total = total + v(i)`
	`  end do`
	`  ...`

11.3.5 Two-Dimensional Arrays

Rows and columns between C and Fortran are switched.

TABLE 11–7 Passing a Two-Dimensional Array

Fortran calls C	C calls Fortran

`REAL Q(10,20)`
`...`
`Q(3,5) = 1.0`
`CALL FIXQ(Q)`
`...`
 	
`----------`
 	
`void fixq_( float a[20][10] )`
`{`
`  ...`
`  a[5][3] = a[5][3] + 1.;`
`  ...`
`}`
	`extern void`
	`  qref_( int[][10], int *);`
	`...`
	`int m[20][10] = ... ;`
	`int sum;`
	`...`
	`qref_( m, &sum );`
	`...`
	
	`----------`
	
	`SUBROUTINE QREF(A,TOTAL)`
	`  INTEGER A(10,20), TOTAL`
	`  DO I = 1,10`
	`  DO J = 1,20`
	`  TOTAL = TOTAL + A(I,J)`
	`  END DO`
	`  END DO`
	`  ...`

11.3.6 Structures

C and Fortran 95 derived types can be passed to each other's routines as long as the corresponding elements are compatible.(**`f95`** accepts legacy **`STRUCTURE`** statements.)

TABLE 11–8 Passing Legacy FORTRAN 77 STRUCTURE Records

Fortran calls C	C calls Fortran

`STRUCTURE /POINT/`
`REAL X, Y, Z`
`END STRUCTURE`
`RECORD /POINT/ BASE`
`EXTERNAL FLIP`
`...`
`CALL FLIP( BASE )`
`...`
 	
`----------`
 	
`struct point {`
`  float x,y,z;`
`};`
`void flip_( struct point *v )`
`{`
`  float t;`
`  t = v -> x;`
`  v -> x = v -> y;`
`  v -> y = t;`
`  v -> z = -2.*(v -> z);`
`}`
	`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`
	`  ...`

Note that Fortran 77 (VAX) structures always have the same alignment as C structures on all platforms. However, the alignment changes between platforms.

TABLE 11–9 Passing Fortran 95 Derived Types

Fortran 95 calls C	C calls Fortran 95

`TYPE point`
`  SEQUENCE`
`  REAL :: x, y, z`
`END TYPE point`
`TYPE (point) base`
`EXTERNAL flip`
`...`
`CALL flip( base)`
`...`
 	
`----------`
 	
`struct point {`
`    float x,y,z;`
`};`
`void flip_( struct point *v )`
`{<`
`    float t;`
`    t = v -> x;`
`    v -> x = v -> y;`
`    v -> y = t;`
`    v -> z = -2.*(v -> z);`
`}`
	`struct point {`
	`  float x,y,z;`
	`};`
	`extern void fflip_ (`
	`     struct point *) ;`
	`...`
	`struct point d;`
	`struct point *ptx = &d;`
	`...`
	`fflip_ (ptx);`
	`...`
	
	`----------`
	
	`SUBROUTINE FFLIP( P )`
	`    TYPE POINT`
	`     SEQUENCE`
	`     REAL :: X, Y, Z`
	`    END TYPE POINT`
	`    TYPE (POINT) P`
	`    REAL :: T`
	`    T = P%X`
	`    P%X = P%Y`
	`    P%Y = T`
	`    P%Z = -2.*P%Z`
	`    ...`

Note that the Fortran 95 standard requires the **SEQUENCE** statement in the definition of the derived type to insure that storage sequence order be preserved by the compiler.

The components of a numeric sequence type are aligned on word (4-byte) boundaries on all platforms by default. This matches the alignment of C structures on x86 platforms, but differs from the alignment of C structures on SPARC platforms. Use the **-aligncommon** option to change the alignment of numeric sequence types to match C structures. Use **-aligncommon=8** to match 32–bit SPARC C structures, **-aligncommon=16** to match 64–bit SPARC.

Derived types not explicitly declared with **SEQUENCE** have the same alignment as C structures on SPARC platforms, but differ on x86 platforms. This alignment cannot be changed with a compiler option.

11.3.7 Pointers

A FORTRAN 77 (Cray) pointer can be passed to a C routine as a pointer to a pointer because the Fortran routine passes arguments by reference.

TABLE 11–10 Passing a FORTRAN 77 (Cray) POINTER

Fortran calls C	C calls Fortran

`REAL X`
`POINTER (P2X, X)`
`EXTERNAL PASS`
`P2X = MALLOC(4)`
`X = 0.`
`CALL PASS(P2X)`
`...`
 	
`----------`
 	
`void pass_(p)`
`> float **p;`
`{`
`  **p = 100.1;`
`}`
	`extern void fpass_( float** );`
	`...`
	`float *p2x;`
	`...`
	` fpass_(&p2x) ;`
	`...`
	
	`----------`
	
	`SUBROUTINE FPASS (P2X)`
	`  REAL X`
	`  POINTER (P2X, X)`
	`  X = 0.`
	`  ...`

C pointers are compatible with Fortran 95 *scalar* pointers, but not *array* pointers.

Fortran 95 calls C with a scalar pointer

Fortran 95 routine:

```
INTERFACE
  SUBROUTINE PASS(P)
   REAL, POINTER :: P
  END SUBROUTINE
END INTERFACE

 REAL, POINTER :: P2X
 ALLOCATE (P2X)
 P2X = 0
 CALL PASS(P2X)
 PRINT*, P2X
END
```

C routine:

```
void pass_(p);
float **p;
{
  **p = 100.1;
}
```

The major difference between Cray and Fortran 95 pointers is that the target of a Cray pointer is always named. In many contexts, declaring a Fortran 95 pointer automatically identifies its target. Also, an explicit **INTERFACE** block is required for the called C routine.

To pass a Fortran 95 pointer to an array or array section requires a specific **INTERFACE** block, as in this example:

Fortran 95 routine:

```
INTERFACE
  SUBROUTINE S(P)
    integer P(*)
  END SUBROUTINE S
END INTERFACE
integer, target:: A(0:9)
integer, pointer :: P(:)
P => A(0:9:2) !! pointer selects every other element of A
call S(P)
...
```

C routine:

```
void s_(int p[])
{
   /* change middle element */
```

```
   p[2] = 444;
}
```

Note that since the C routine S is not a Fortran 95 routine, you cannot define it to be assumed shape (**integer P(:)**) in the interface block. If the C routine needs to know the actual size of the array it must be passed as an argument to the C routine.

Again, keep in mind that subscripting between C and Fortran differs in that C arrays start at subscript 0.

11.4 Passing Data Arguments by Value

Fortran 95 programs should use the **VALUE** attribute in dummy arguments when being called from C, and supply an **INTERFACE** block for C routines that are called from Fortran 95.

TABLE 11–11 Passing Simple Data Elements Between C and Fortran 95

Fortran 95 calls C	C calls Fortran 95
PROGRAM callc **INTERFACE** **INTEGER FUNCTION crtn(I)** **BIND(C) crtn** **INTEGER, VALUE, INTENT(IN) :: I** **END FUNCTION crtn** **END INTERFACE** **M = 20** **MM = crtn(M)** **WRITE (*,*) M, MM** **END PROGRAM** **--------------------------------** **int crtn(int x)** **{** **int y;** **printf("%d input \n", x);** **y = x + 1;** **printf("%d returning \n",y);** **return(y);** **}** **--------------------------------** *Results:* 20 input 21 returning 20 21	**#include <stdlib.h>** **int main(int ac, char *av[]) {** **to_fortran_(12);** **}** **--------------------------------** **SUBROUTINE to_fortran(i)** **INTEGER, VALUE :: i** **PRINT *, i** **END**

Note that if the C routine will be called with different data types as an actual argument, you should include a **!$PRAGMA IGNORE_TKR I** in the interface block to inhibit the compiler from requiring a match in type, kind, and rank between the actual and dummy argument.

With legacy Fortran 77, call by value was available only for simple data, and only by Fortran 77 routines calling C routines. There was no way for a C routine to call a Fortran 77 routine and pass arguments by value. Arrays, character strings, or structures are best passed by reference.

To pass a value to a C routine from a Fortran 77 routine, use the nonstandard Fortran function **%VAL(***arg***)** as an argument in the call.

In the following example, the Fortran 77 routine passes x by value and y by reference. The C routine incremented both x and y, but only y is changed.

Fortran calls C

Fortran routine:

```
REAL x, y
x = 1.
y = 0.
PRINT *, x,y
CALL value( %VAL(x), y)
PRINT *, x,y
END
```

C routine:

```
void value_( float x, float *y)
{
  printf("%f, %f\n",x,*y);
  x = x + 1.;
  *y = *y + 1.;
  printf("%f, %f\n",x,*y);
}
```

Compiling and running produces output:

```
1.00000 0.          x and y from Fortran
1.000000, 0.000000  x and y from C
2.000000, 1.000000  new x and y from C
1.00000 1.00000     new x and y from Fortran
```

11.5 Functions That Return a Value

A Fortran function that returns a value of type BYTE , INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or REAL*16 is equivalent to a C function that returns a compatible type (see Table 11–1). There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

11.5.1 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 11–12 Functions Returning a REAL or Float Value

Fortran calls C	C calls Fortran

`real ADD1, R, S`
`external ADD1`
`R = 8.0`
`S = ADD1( R )`
`...`
 	
`----------`
 	
`float add1_( pf )`
`float *pf;`
`{`
`  float f ;`
`  f = *pf;`
`  f++;`
`  return ( f );`
`}`
	`float r, s;`
	`extern float fadd1_() ;`
	`r = 8.0;`
	`s = fadd1_( &r );`
	`...`
	
	`----------`
	
	`real function fadd1 (p)`
	`  real p`
	`  fadd1 = p + 1.0`
	`  return`
	`end`

11.5.2 Returning COMPLEX Data

The situation for interoperability of COMPLEX data differs between 32-bit implementations and 64-bit SPARC implementations.

11.5.2.1 32-bit Platforms

A Fortran function returning COMPLEX or DOUBLE COMPLEX on 32-bit 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:

Fortran function	C function
`COMPLEX FUNCTION CF(` *a1,a2,...,an)*	`cf_ (`*return, a1, a2, ..., an)* `struct { float r, i; } *`*return*

TABLE 11–13 Function Returning COMPLEX Data (32–bit SPARC)

Fortran calls C	C calls Fortran

`COMPLEX U, V, RETCPX`
`EXTERNAL RETCPX`
`U = ( 7.0, -8.0)`
`V = RETCPX(U)`
`...`
 	
`----------`
 	
`struct complex { float r, i; };`
`void retcpx_( temp, w )`
`struct complex *temp, *w;`
`{`
`  temp->r = w->r + 1.0;`
`  temp->i = w->i + 1.0;`
`  return;`
`}`
	`struct complex { float r, i; };`
	`struct complex c1, c2;<`
	`struct complex *u=&c1, *v=&c2;`
	`extern retfpx_();`
	`u -> r = 7.0;`
	`u -> i = -8.0;`
	`retfpx_( v, u );`
	`...`
	
	`----------`
	
	`COMPLEX FUNCTION RETFPX(Z)`
	`  COMPLEX Z`
	`  RETFPX = Z + (1.0, 1.0)`
	`  RETURN`
	`END`

11.5.2.2 64–bit SPARC Platforms

In 64-bit SPARC environments, **`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`**. For 64–bit SPARC, a C function returning a structure whose fields are all floating-point types will return the structure in the floating-point registers if at most 4 such registers are needed to do so.The general pattern for the Fortran function and its corresponding C function on 64–bit SPARC platforms is:

Fortran function	C function
`COMPLEX FUNCTION CF(`*a1, a2, ..., an)*	**`struct {float r,i;} cf_ (`***a1, a2, ..., an)*

TABLE 11–14 Function Returning COMPLEX Data (64–bit SPARC)

Fortran calls C

```
COMPLEX U, V, RETCPX
EXTERNAL RETCPX
U = ( 7.0, -8.0)
V = RETCPX(U)
...

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

struct complex {float r, i; };
struct complex retcpx_(struct complex *w )
{
    struct complex temp;
    temp.r = w->r + 1.0;
    temp.ii = w->i + 1.0;
    return (temp);
}
```

C calls Fortran

```
struct complex { float r, i; };
struct complex c1, c2;
struct complex *u=&c1;
extern struct complex retfpx_(struct complex *);
  u -> r = 7.0;
  u -> i = -8.0;
  retfpx_( u );
...

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

COMPLEX FUNCTION RETFPX(Z)
  COMPLEX Z
  RETFPX = Z + (1.0, 1.0)
  RETURN
END
```

11.5.3 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:

Fortran function	C function
CHARACTER*n* FUNCTION C(***a1, ..., an***)**	**void c_ (***result, length, a1, ..., an***)** **char** *result***[ ];** **long length;**

Here is an example:

TABLE 11–15 A Function Returning a CHARACTER String

Fortran calls C	C calls Fortran
CHARACTER STRING*16, CSTR*9 **STRING = ' '** **STRING = '123' // CSTR('*',9)** **...** **----------** **void cstr_(char *p2rslt,** **long rslt_len,** **char *p2arg,** **int *p2n,** **long arg_len)** **{ /* return n copies of arg */** **int count, i;** **char *cp;** **count = *p2n;** **cp = p2rslt;** **for (i=0; i<count; i++) {** ***cp++ = *p2arg ;** **}** **}**	**void fstr_(char *, long,** **char *, int *, long);** **char sbf[9] = "123456789";** **char *p2rslt = sbf;** **int rslt_len = sizeof(sbf);** **char ch = '*';** **int n = 4;** **int ch_len = sizeof(ch);** **/* make n copies of ch in sbf** ***/** **fstr_(p2rslt, rslt_len,** **&ch, &n, ch_len);** **...** **----------** **FUNCTION FSTR(C, N)** **CHARACTER FSTR*(*), C** **FSTR = ''** **DO I = 1,N** **FSTR(I:I) = C** **END DO** **FSTR(N+1:N+1) = CHAR(0)** **END**

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. Fortran strings are not null-terminated by default.

11.6 Labeled COMMON

Fortran labeled COMMON can be emulated in C by using a global **struct**.

TABLE 11–16 Emulating Labeled COMMON

Fortran COMMON Definition	C "COMMON" Definition

`COMMON /BLOCK/ ALPHA,NUM`
`...`
	`extern struct block {`
	`  float alpha;`
	`  int num;`
	`};`
	`extern struct block block_ ;`
	`main ()`
	`{`
	`...`
	`  block_.alpha = 32.;`
	`  block_.num += 1;`
	`...`
	`}`

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.

f95 aligns data in common blocks to at most 4-byte boundaries by default. To obtain the natural alignment for all data elements inside a common block and match the default structure alignment, use **-aligncommon=16** when compiling the Fortran routines.

11.7 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.

Remember: even though the main program is in C, you should link with **f95**.

11.8 Alternate Returns

Fortran 77's alternate returns mechanism is obsolete 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. Fortran 95 will accept Fortran 77 alternate returns, but its use should be discouraged.

The implementation returns the **int** value of the expression on the RETURN statement. This is implementation dependent and its use should be avoided.

TABLE 11–17 Alternate Returns

C calls Fortran	Running the Example

`int altret_ ( int * );`
`main ()`
`{`
`  int k, m ;`
`  k =0;`
`  m = altret_( &k ) ;`
`  printf( "%d %d\n", k, m);`
`}`
 	
`----------`
 	
`SUBROUTINE ALTRET( I, *, *)`
`  INTEGER I`
`  I = I + 1`
`  IF(I .EQ. 0) RETURN 1`
`  IF(I .GT. 0) RETURN 2`
`  RETURN`
`END`
	`demo% cc -c tst.c`
	`demo% f95 -o alt alt.f tst.o`
	`alt.f:`
	`altret:`
	`demo% alt`
	`1 2`
	
	The C routine receives the return value 2 from the Fortran routine because it executed the RETURN 2 statement.

11.9 Fortran 2003 Interoperability With C

The Fortran 2003 draft standard (available from `http://www.j3-fortran.org`) provides a means of referencing procedures and global variables defined by the C programming language from within a Fortran 95 program. And, conversely, provides a means for defining Fortran subprograms or global variables so that they can be referenced from C procedures.

By design, use of these features to accomplish interoperability between Fortran 95 and C programs insures portability across standards-conforming platforms.

Fortran 2003 provides the **BIND** attribute for derived types, and the **ISO_C_BINDING** intrinsic module. The module makes accessible to the Fortran program certain named constants, derived types, and procedures that support specification of interoperable objects. The details can be found in the Fortran 2003 standard.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z