Fortran User's Guide

Forte Developer 6 update 2
(Sun WorkShop 6 update 2)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-7988-10
July 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Important Note on New Product Names

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

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

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

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

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

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

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

Contents

Tables

Before You Begin

The *Fortran User's Guide* describes the compile-time environment and command-line options for the Sun WorkShop™ 6 Fortran compilers: `f77` (FORTRAN 77) and `f95` (Fortran 95).

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

Discussion of Fortran programming issues on Solaris™ operating environments, including input/output, application development, library creating and use, program analysis, porting, optimization, and parallelization can be found in the companion Sun WorkShop *Fortran Programming Guide*.

Other Fortran manuals in this collection include the *Fortran Library Reference*, and the *FORTRAN 77 Language Reference*. See "Accessing Related Documentation" on page 6.

How This Book Is Organized

Chapter 1 briefly describes the features of the compilers.

Chapter 2 discusses the compiler environments.

Chapter 3 gives detailed descriptions of all the compile-time command-line options and flags.

Appendix A lists error messages issued by the Fortran runtime library and operating environment.

Appendix B notes new features of the compilers and changes in recent releases.

Appendix C describes the differences between the Sun `f95` compiler and the Fortran 95 standard, and incompatibilities with `f77` programs.

Appendix D lists all the platform system names accepted by the compiler `-xtarget` option.

Appendix E summarizes the directives accepted by the compilers, including parallelization and OpenMP directives.

Typographic Conventions

The following table and notes describe the typographical conventions used in the manual.

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*.

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

```
∆∆36.001
```

- FORTRAN 77 examples appear in tab format, while Fortran 95 examples appear in free format. Examples common to both Fortran 77 and 95 use tab format except where indicated.
- The FORTRAN 77 standard uses an older convention of spelling the name "FORTRAN" capitalized. Sun documentation uses both FORTRAN and Fortran. The current convention is to use lower case: "Fortran 95".

- References to online man pages appear with the topic name and section number. For example, a reference to GETENV will appear as `getenv`(3F), implying that the man command to access this page would be: `man -s 3F getenv`
- System Administrators may install the Sun WorkShop Fortran compilers and supporting material at: <*install_point*>`/SUNWspro/` where <*install_point*> is usually `/opt` for a standard install. This is the location assumed in this book.

Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Supported Platforms

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

Accessing Sun WorkShop Development Tools and Man Pages

The Sun WorkShop product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Sun WorkShop compilers and tools, you must have the Sun WorkShop component directory in your `PATH` environment variable. To access the Sun WorkShop man pages, you must have the Sun WorkShop man page directory in your `MANPATH` environment variable.

For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this release, see the *Sun WorkShop 6 update 2 Installation Guide* or your system administrator.

Note – The information in this section assumes that your Sun WorkShop 6 update 2 products are installed in the /opt directory. If your product software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

Accessing Sun WorkShop Compilers and Tools

Use the steps below to determine whether you need to change your PATH variable to access the Sun WorkShop compilers and tools.

To Determine If You Need to Set Your PATH Environment Variable

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

```
% echo $PATH
```

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

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

To Set Your PATH Environment Variable to Enable Access to Sun WorkShop Compilers and Tools

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**

2. **Add the following to your PATH environment variable.**

   /opt/SUNWspro/bin

Accessing Sun WorkShop Man Pages

Use the following steps to determine whether you need to change your MANPATH variable to access the Sun WorkShop man pages.

To Determine If You Need to Set Your MANPATH Environment Variable

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

```
% man workshop
```

2. **Review the output, if any.**

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

To Set Your MANPATH Environment Variable to Enable Access to Sun WorkShop Man Pages

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**
2. **Add the following to your MANPATH environment variable.**

   /opt/SUNWspro/man

Accessing Sun WorkShop Documentation

You can access Sun WorkShop product documentation at the following locations:

- **The product documentation is available from the documentation index installed with the product on your local system or network.**

  Point your Netscape™ Communicator 4.0 or compatible Netscape version browser to the following file:

  /opt/SUNWspro/docs/index.html

If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- **Manuals are available from the docs.sun.com[sm] Web site.**

  The `docs.sun.com` Web site (`http://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 installed with the product on your local system or network.

Accessing Related Documentation

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

Document Collection	Document Title	Description
Forte™ for High Performance Computing Collection	*Fortran Programming Guide*	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran Library Reference	Provides details about the library routines supplied with the Fortran compilers
	FORTRAN 77 Language Reference	Provides a complete language reference to Sun FORTRAN 77.
Numerical Computation Guide Collection	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Solaris 8 Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris 8 Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris 8 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.

Ordering Sun Documentation

You can order product documentation directly from Sun through the `docs.sun.com` Web site or from Fatbrain.com, an Internet bookstore. You can find the Sun Documentation Center on Fatbrain.com at the following URL:

`http://www.fatbrain.com/documentation/sun`

Sending Your Comments

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

`docfeedback@sun.com`

CHAPTER 1

Introduction

The Sun Fortran compilers, `f77` and `f95`, described in this book (and the companion Sun WorkShop *Fortran Programming Guide*) are available under the Solaris operating environment on SPARC and UltraSPARC™ platforms. The compilers themselves conform to published Fortran language standards, and provide many extended features, including multiprocessor parallelization, sophisticated optimized code compilation, and mixed C/Fortran language support.

The Fortran compilers are components of the Forte™ for High Performance Computing (Sun Performance WorkShop™) software. The Fortran 90 compiler, `f90`, has been renamed Fortran 95, `f95`. The `f90` command is now an alias for `f95` — both invoke the Fortran 95 compiler.

Standards Conformance

- `f77` was designed to be compatible with the ANSI X3.9-1978 Fortran standard and the corresponding International Organization for Standardization (ISO) 1539-1980, as well as standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- `f95` was designed to be compatible with the ANSI X3.198-1992, ISO/IEC 1539:1991, and ISO/IEC 1539:1997 standards documents.
- Floating-point arithmetic for both compilers is based on IEEE standard 754-1985, and international standard IEC 60559:1989.
- On SPARC platforms, both compilers provide support for the optimization-exploiting features of SPARC V8, and SPARC V9, including the UltraSPARC implementation. These features are defined in the SPARC Architecture Manuals, Version 8 (ISBN 0-13-825001-4), and Version 9 (ISBN 0-13-099227-5), published by Prentice-Hall for SPARC International.
- In this document, “Standard” means conforming to the versions of the standards listed above. “Non-standard” or “Extension” refers to features that go beyond these versions of these standards.

The responsible standards bodies may revise these standards from time to time. The versions of the applicable standards to which these compilers conform may be revised or replaced, resulting in features in future releases of the Sun Fortran compilers that create incompatibilities with earlier releases.

Features of the Fortran Compilers

Sun Fortran compilers provide the following features or extensions:

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

Note – Parallelization features of the Fortran compilers require a Forte for HPC license.

- `f77`: Many VAX/VMS Fortran 5.0 extensions, including:
  - `NAMELIST`
  - `DO WHILE`
  - Structures, records, unions, maps
  - Variable format expressions
  - Recursion
  - Pointers
  - Double-precision complex
  - Quadruple-precision real
  - Quadruple-precision complex
- Cray-style parallelization directives, including `TASKCOMMON`, with extensions for `f95`.
- OpenMP parallelization directives accepted by `f95.`
- Global, peephole, and potential parallelization optimizations produce high performance applications. Benchmarks show that optimized applications can run significantly faster when compared to unoptimized code.
- Common calling conventions on Solaris systems permit routines written in C or C++ to be combined with Fortran programs.
- Support for 64-bit enabled Solaris environments on UltraSPARC platforms.
- Call-by-value, `%VAL`, implemented in both `f77` and `f95`.
- Interoperability between Fortran 77 and Fortran 95 programs and object binaries.
- Interval Arithmetic expressions in `f95`.

See Appendix B for details on new and extended features added to the compilers with each software release.

Other Fortran Utilities

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

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

Debugging Utilities

The following debugging utilities are available:

- **`error`** — (*f77 only*) A utility to merge compiler error messages with the Fortran source file. (This utility is included if you do a developer install, rather than an end user install of Solaris; it is also included if you install the `SUNWbtool` package.)
- **`-Xlist`** — A compiler option to check across routines for consistency of arguments, COMMON blocks, and so on.
- **Sun WorkShop** —Provides a visual debugging environment based on `dbx` and includes a data visualizer and performance data collector.

Sun Performance Library™

The Sun Performance Library is a library of optimized subroutines and functions for computational linear algebra and Fourier transforms. It is based on the standard libraries LAPACK, 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 Fortran and C* for details. (Man pages for the performance library routines are in section 3P.)

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.

Man Pages

Online manual (`man`) pages provide immediate documentation about a command, function, subroutine, or collection of such things. See the Preface for the proper setting of the `MANPATH` environment variable for accessing Sun WorkShop man pages.)

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:

`f77`(1) and `f95`(1)	The Fortran compilers command-line options
`analyzer`(1)	Sun WorkShop 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
`fsplit`(1)	Pre-processor splits Fortran 77 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

READMEs

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

TABLE 1-1 READMEs of Interest

README File	Describes...
`fortran_77`	new and changed features, known limitations, documentation errata for this release of the FORTRAN 77 compiler, `f77`.
`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
`interval_arithmetic`	overview of the interval arithmetic features in `f95`
`math_libraries`	optimized and specialized math libraries available.
`omp_directives.pdf`	summarizes OpenMP directives accepted by `f95`. (This is a PDF file.)
`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.
`64bit_Compilers`	compiling for 64-bit Solaris operating environments.
`performance_library`	overview of the Sun Performance Library

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

```
f95 -xhelp=readme
```

will display the `fortran_95` README file directly.

Command-Line Help

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

```
%f77 -help  -or-
 f95 -help
_____________________________________________________________________
Items within [ ] are optional. Items within < > are variable
parameters.Bar | indicates choice of literal values. For example:
   -someoption[=<yes|no>] implies  -someoption  is
   -someoption=yes
_____________________________________________________________________
-a:             Collect data for tcov basic block profiling
                    (old format)
-ansi:          Report non-ANSI extensions.
-arg=local:     Preserve actual arguments over ENTRY statements
-autopar:       Enable automatic loop parallelization
                        (requires WorkShop license)
-Bdynamic:      Allow dynamic linking
-Bstatic:       Require static linking
-c:             Compile only - produce .o files, suppress linking
-C:             Enable runtime subscript range checking
-cg89:          Generate code for generic SPARC V7 architecture
-cg92:          Generate code for SPARC V8 architecture
-copyargs:      Allow assignment to constant arguments
...etc.
```

CHAPTER 2

Using Sun Fortran Compilers

This chapter describes how to use the Fortran 77 and Fortran 95 compilers.

The principal use of any compiler is to transform a program written in a procedural language like Fortran into a data file that is executable by the target computer hardware. As part of its job, the compiler may also automatically invoke a system linker to generate the executable file.

The Sun Fortran 77 and Fortran 95 compilers can also be used to:

- Generate a parallelized executable file for multiple processors (`-parallel`).
- Analyze program consistency across source files and subroutines and generate a report (`-Xlist`).
- Transform source files into:
  - Relocatable binary (`.o`) files, to be linked later into an executable file or static library (`.a`) file.
  - A dynamic shared library (`.so`) file (`-G`).
- Link files into an executable file.
- Compile an executable file with runtime debugging enabled (`-g`).
- Compile with runtime statement or procedure level profiling (`-pg`).
- Compile an executable file with runtime parallelized loop profiling (`-Zlp`).
- Check source code for ANSI standards conformance (`-ansi`).

A Quick Start

This section provides a quick overview of how to use the Sun Fortran compilers to compile and run Fortran programs. A full reference to command-line options appears in the next chapter.

Note – The command line examples in this chapter primarily show `f77` usages. Except where noted, equivalent usages of `f95` are similarly valid; however, the printed output may be slightly different.

The very basic steps to running a Fortran application involve using an editor to create a Fortran source file with a `.f`, `.for`, `.f90`, `.f95`, `.F`, `.F90`, or `.F95` filename suffix; invoking the compiler to produce an executable; and finally, launching the program into execution by typing the name of the file:

Example: This program displays a message on the screen:

```
demo% cat greetings.f
      PROGRAM GREETINGS
      PRINT *, 'Real programmers write Fortran!'
      END
demo% f77 greetings.f
greetings.f:
  MAIN greetings:
demo% a.out
 Real programmers write Fortran!
demo%
```

In this example, `f77` compiles source file `greetings.f` and links the executable program onto the file, `a.out`, by default. To launch the program, the name of the executable file, `a.out`, is typed at the command prompt.

Traditionally, UNIX compilers write executable output to the default file called `a.out`. It can be awkward to have each compilation write to the same file. Moreover, if such a file already exists, it will be overwritten by the next run of the compiler. Instead, use the `-o` compiler option to explicitly specify the name of the executable output file:

```
demo% f77 –o greetings greetings.f
greetings.f:
MAIN greetings:
demo%
```

In the preceding example, the `-o` option tells the compiler to write the executable code to the file `greetings`. (By convention, executable files usually are given the same name as the main source file, but without an extension.)

Alternatively, the default `a.out` file could be renamed via the `mv` command after each compilation. Either way, run the program by typing the name of the executable file:

```
demo% greetings
 Real programmers write Fortran!
demo%
```

Here is the same example, using `f95`:

```
demo% cat greetings.f95
program greetings
print*, 'Real programmers write Fortran 95!'
end
demo% f95 -o greetings greetings.f95
demo% greetings
 Real programmers write Fortran 95!
demo%
```

The next sections of this chapter discuss the conventions used by the `f77` and `f95` commands, compiler source line directives, and other issues concerning the use of these compilers. The next chapter describes the command-line syntax and all the options in detail.

Invoking the Compiler

The syntax of a *simple* compiler command invoked at a shell prompt is:

`f77` *[options] files... invokes the Fortran 77 compiler*

`f95` *[options] files... invokes the Fortran 95 compiler*

Here *files...* is one or more Fortran source file names ending in `.f`, `.F`, `.f90`, `.f95`, `.F90`, `.F95`, or `.for`; *options* is one or more of the compiler option flags. (Files with names ending in a `.f90` or `.f95` extension are "free-format" Fortran 95 source files recognized only by the `f95` compiler.)

In the example below, `f95` is used to compile two source files to produce an executable file named `growth` with runtime debugging enabled:

```
demo% f95 -g -o growth growth.f fft.f95
```

Note – You can invoke the Sun WorkShop 6 Fortran 95 compiler with either the **`f95`** or **`f90`** command — **`f90`** is now an alias for **`f95`**.

Compile-Link Sequence

In the previous example, the compiler automatically generates the loader object files, `growth.o` and `fft.o`, and then invokes the system linker to create the executable program file `growth`.

After compilation, the object files, `growth.o` and `fft.o`, will remain. This convention permits easy relinking and recompilation of files.

If the compilation fails, you will receive a message for each error. No `.o` files are generated for those source files with errors, and no executable program file is written.

Command-Line File Name Conventions

The suffix extension attached to file names appearing on the command-line determine how the compiler will process the file. File names with a suffix extension other than one of those listed below, or without an extension, are passed to the linker.

TABLE 2-1 File Name Suffixes Recognized by Sun Fortran Compilers

Suffix	Language	Action
`.f`	Fortran 77 or Fortran 95 fixed-format	Compile Fortran source files, put object files in current directory; default name of object file is that of the source but with `.o` suffix.
`.f95` `.f90`	Fortran 95 free-format	Same action as `.f` (*f95 only)*
`.for`	Fortran 77 or Fortran 95	Same action as `.f`.

TABLE 2-1 File Name Suffixes Recognized by Sun Fortran Compilers *(Continued)*

Suffix	Language	Action
`.F`	Fortran 77 or Fortran 95 fixed-format	Apply the Fortran (or C) preprocessor to the Fortran 77 source file before compilation.
`.F95` `.F90`	Fortran 95 free-format	Apply the Fortran (or C) preprocessor to the Fortran 95 free-format source file before Fortran compiles it. (*f95 only)*
`.s`	Assembler	Assemble source files with the assembler.
`.S`	Assembler	Apply the C preprocessor to the assembler source file before assembling it.
`.il`	Inline expansion	Process template files for inline expansion. The compiler will use templates to expand inline calls to selected routines. (Template files are special assembler files; see the `inline`(1) man page.)
`.o`	Object files	Pass object files through to the linker.
`.a`, `.s` `.o`, `.so.`*n*	Libraries	Pass names of libraries to the linker. `.a` files are static libraries, `.so` and `.so.`*n* files are dynamic libraries.

Fortran 95 free-format is described in Appendix C of this manual.

Source Files

The Fortran compilers will accept multiple source files on the command line. A single source file, also called a *compilation unit*, may contain any number of procedures (main program, subroutine, function, block data, module, and so on). Applications may be configured with one source code procedure per file, or by gathering procedures that work together into single files. The *Fortran Programming Guide* describes the advantages and disadvantages of these configurations.

Source File Preprocessors

Both `f77` and `f95` support two source file preprocessors, `fpp` and `cpp`. Either can be invoked by the compiler to expand source code "macros" and symbolic definitions prior to compilation. The compilers will use `fpp` by default; the `-xpp=cpp` option changes the default from `fpp` to `cpp`. (See also the discussion of the `-D`*name* option).

`fpp` is a Fortran-specific source preprocessor. See the `fpp`(1) man page and the `fpp` README for details. It is invoked by default by `f77` on files with a `.F` extension and by `f95` on files with a `.F`, `.F90`, or `.F95` extension.

The source code for `fpp` is available from the Netlib web site at

```
http://www.netlib.org/fortran/
```

See `cpp`(1) for information on the standard Unix C language preprocessor. Use of `fpp` over `cpp` is recommended on Fortran source files.

Separate Compiling and Linking

You can compile and link in separate steps. The `-c` option compiles source files and generates `.o` object files, but does not create an executable. Without the `-c` option the compiler will invoke the linker. By splitting the compile and link steps in this manner, a complete recompilation is not needed just to fix one file, as shown in the following example:

Compile one file and link with others in separate steps:

```
demo% f95 -c file1.f                              (Make new object file)
demo% f95 -o prgrm file1.o file2.o file3.o        (Make executable file)
```

Be sure that the link step lists *all* the object files needed to make the complete program. If any object files are missing from this step, the link will fail with undefined external reference errors (missing routines).

Consistent Compiling and Linking

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

`–a, –autopar, -B`*x*`, –fast, -G, -L`*path*`, -l`*name*`, -mt, -xmemalign, -nolib, -norunpath, –p, –pg, -xlibmopt, -xlic_lib=`*name*`, -xprofile=`*p*

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

```
demo% f95 -c -a sbr.f
demo% f95 -c smain.f
demo% f95 -a sbr.o smain.o          link step; passes -a to the linker
```

Also, a number of options require that *all* source files be compiled with that option, *including* the link step. These include:

`-autopar, -aligncommon, -d`*x*`, -dalign, -dbl, -explicitpar, -f, -misalign, -native, -parallel, -r8, -xarch=`*a*`, -xcache=`*c*`, -xchip=`*c*`, -xF, -xtarget=`*t*`, -xtypemap, -ztext`

Linking Mixed Fortran 95 and Fortran 77 Compilations

As a general rule, if *any* of the object files that make up a program were compiled with `f95`, then the final link step must be done with `f95`. Use `f77` to produce the executable file only if *none* of the `.o` object files were compiled with `f95`. See also Appendix C, "Compatibility with FORTRAN 77" on page 163.

Unrecognized Command-Line Arguments

Any arguments on the command-line that the compiler does not recognize are interpreted as being possibly linker options, object program file names, or library names.

The basic distinctions are:

- Unrecognized *options* (with a `-`) generate warnings.
- Unrecognized *non-options* (no `-`) generate no warnings. However, they are passed to the linker and if the linker does not recognize them, they generate linker error messages.

For example:

```
demo% f95 -bit move.f            <- -bit is not a recognized f95 option
f95: Warning: Option -bit passed to ld, if ld is invoked, ignored
otherwise
demo% f95 fast move.f            <- The user meant to type -fast
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors.  No output written to a.out
```

Note that in the first example, `-bit` is not recognized by `f95` and the option is passed on to the linker (`ld`), who tries to interpret it. Because single letter `ld` options may be strung together, the linker sees `-bit` as `-b -i -t`, which are all legitimate `ld` options! This may (or may not) be what the user expects, or intended.

In the second example, the user intended to type the `f77/f95` option `-fast` but neglected the leading dash. The compiler again passes the argument to the linker which, in turn, interprets it as a file name.

These examples indicate that extreme care should be observed when composing compiler command lines!

Modules (Fortran 95)

`f95` automatically creates module interface files for each `MODULE` declaration encountered in the source files, and searches for modules referenced by a `USE` statement. For each module encountered (`MODULE` *module_name*), the compiler generates a corresponding file, *module_name*`.mod`, in the current directory. For example, `f95` generates the module information file `list.mod` for the `MODULE list` unit found on file `mysrc.f95` .

The compiler searches the current directory for module files referenced in `USE` statements. Module files must be compiled before compiling any source file referencing a `MODULE` in a `USE` statement. Directories can be added to the search path with the `-M` command-line option. However, individual `.mod` files cannot be specified directly on the command line.

The `f95` compiler also creates an object file *filename*`.o` for every source file containing `MODULE` statement, and these implementation object files must be included when linking to create an executable. See page 167.

Directives

Use a source code *directive*, a form of Fortran comment, to pass specific information to the compiler regarding special optimization or parallelization choices. Compiler directives are also sometimes called *pragmas.* The compilers recognize a set of general directives and parallelization directives. Fortran 95 also processes OpenMP shared memory multiprocessing directives.

Directives unique to `f95` are described in Appendix C. A complete summary of all the directives recognized by `f77` and `f95` appears in Appendix E.

Note – Directives are not part of the Fortran standard.

General Directives

The various forms of a general Sun Fortran directive are:

```
C$PRAGMA keyword ( a [ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…
C$PRAGMA  SUN keyword ( a [ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…
C$PRAGMA  SPARC keyword ( a [ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…
```

The variable *keyword* identifies the specific directive. Additional arguments or suboptions may also be allowed. (Some directives require the additional keyword `SUN` or `SPARC`, as shown above.)

A general directive has the following syntax:

- In column one, any of the comment-indicator characters `c`, `C`, `!`, or `*`
- For `f95` free-format, `!` is the only comment-indicator recognized (`!$PRAGMA`). The examples in this chapter assume fixed-format.
- The next seven characters are `$PRAGMA`, no blanks, in either uppercase or lowercase
- With `f77`, directives using the `!` comment-indicator character may appear in any position on the line. With `f95`, this is only possible for free-format source programs.

Observe the following restrictions:

- After the first eight characters, blanks are ignored, and uppercase and lowercase are equivalent, as in Fortran text.
- Because it is a comment, a directive cannot be continued, but you can have many `C$PRAGMA` lines, one after the other, as needed.
- If a comment satisfies the above syntax, it is expected to contain one or more directives recognized by the compiler; if it does not, a warning is issued.
- The C preprocessor, `cpp`, will expand macro symbol definitions within a comment or directive line; the Fortran preprocessor, `fpp`, will not expand macros in comment lines. `fpp` will recognize legitimate `f77` and `f95` directives and allow limited substitution outside directive keywords. However, be careful with directives requiring the keyword **`SUN`**. `cpp` will replace lower-case **`sun`** with a predefined value. Also, if you define a `cpp` macro **`SUN`**, it might interfere with the **`SUN`** directive keyword. A general rule would be to spell those pragmas in mixed case if the source will be processed by `cpp` or `fpp`, as in:

  ```
  C$PRAGMA Sun UNROLL=3
  ```

The Fortran compilers recognize the following general directives:

TABLE 2-2 Summary of General Fortran Directives

`C` *Directive*	`C$PRAGMA C(`*list*`)` Declares a list of names of external functions as C language routines.
`UNROLL` *Directive*	`C$PRAGMA SUN UNROLL=`*n* Advises the compiler that the following loop can be unrolled to a length *n*.
`WEAK` *Directive*	`C$PRAGMA WEAK(`*name*`[=`*name2*`])` Declares *name* to be a weak symbol, or an alias for *name2*.
`OPT` *Directive*	`C$PRAGMA SUN OPT=`*n* Set optimization level for a subprogram to *n*.
`PIPELOOP` *Directive*	`C$PRAGMA SUN PIPELOOP=`*n* Assert dependency in the following loop exists between iterations *n* apart.
`NOMEMDEP` *Directive*	`C$PRAGMA SUN NOMEMDEP` Assert there are no memory dependencies in the following loop.

`PREFETCH` *Directives*	`C$PRAGMA SPARC_PREFETCH_READ_ONCE(`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_READ_MANY(`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_WRITE_ONCE(`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_WRITE_MANY(`*name*`)`
	
	Request compiler generate prefetch instructions for references to name. (Requires `-xprefetch` option.)

The C Directive

The `C()` directive specifies that its arguments are external functions. It is equivalent to an `EXTERNAL` declaration except that unlike ordinary external names, the Fortran compiler will not append an underscore to these argument names. See the C-Fortran Interface chapter in the *Fortran Programming Guide* for more details.

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

Example - compiling ABC and XYZ for C:

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

The UNROLL Directive

The UNROLL directive requires that you specify SUN after C$PRAGMA.

The C$PRAGMA SUN UNROLL=*n* directive instructs the compiler to unroll the following loop *n* times during its optimization pass. (The compiler will unroll a loop only when its analysis regards such unrolling as appropriate.)

n is a positive integer. The choices are:

- If *n*=1, the optimizer *may not* unroll any loops.
- If *n*>1, the optimizer *may* unroll loops *n* times.

If any loops are actually unrolled, the executable file becomes larger. For further information, see the *Fortran Programming Guide* chapter on performance and optimization.

Example - unrolling loops two times:

```
C$PRAGMA SUN UNROLL=2
```

The WEAK Directive

The WEAK directive defines a symbol to have less precedence than an earlier definition of the same symbol. This pragma is used mainly in sources files for building libraries. The linker does not produce an error message if it is unable to resolve a weak symbol.

```
C$PRAGMA WEAK (name1 [=name2])
```

WEAK (*name1*) defines *name1* to be a weak symbol. The linker does not produce an error message if it does not find a definition for *name1*.

WEAK (*name1*=*name2*) defines *name1* to be a weak symbol and an alias for *name2*.

If your program calls but does not define *name1*, the linker uses the definition from the library. However, if your program defines its own version of *name1*, then the program's definition is used and the weak global definition of *name1* in the library is

not used. If the program directly calls *name2*, the definition from library is used; a duplicate definition of *name2* causes an error. See the Solaris *Linker and Libraries Guide* for more information.

The OPT Directive

The OPT directive requires that you specify SUN after C$PRAGMA.

The OPT directive sets the optimization level for a subprogram, overriding the level specified on the compilation command line. The directive must appear immediately before the target subprogram, and only applies to that subprogram. For example:

```
C$PRAGMA SUN OPT=2
        SUBROUTINE smart(a,b,c,d,e)
        ...etc
```

When the above is compiled with an f77 command that specifies -O4, the directive will override this level and compile the subroutine at -O2. Unless there is another directive following this routine, the next subprogram will be compiled at -O4.

The routine must also be compiled with the -xmaxopt[=*n*] option for the directive to be recognized. This compiler option specifies a maximum optimization value for PRAGMA OPT directives: if a PRAGMA OPT specifies an optimization level greater than the -xmaxopt level, the -xmaxopt level is used.

The NOMEMDEP Directive

The NOMEMDEP directive requires that you specify SUN after C$PRAGMA.

This directive must appear immediately before a DO loop. It asserts to the optimizer that there are no memory-based dependencies within an iteration of the loop to inhibit parallelization. Requires -parallel or -explicitpar options..

The PIPELOOP=*n* Directive

The PIPELOOP=*n* directive requires that you specify SUN after C$PRAGMA.

This directive must appear immediately before a DO loop. *n* is a positive integer constant, or zero, and asserts to the optimizer a dependence between loop iterations. A value of zero indicates that the loop has no inter-iteration (loop-carried)

dependencies and can be freely pipelined by the optimizer. A positive *n* value implies that the I-th iteration of the loop has a dependency on the (I-*n*)-th iteration, and can be pipelined at best for only *n* iterations at a time.

```
C    We know that the value of K is such that there can be no
C    cross-iteration dependencies (E.g. K>N)
C$PRAGMA SUN PIPELOOP=0
      DO I=1,N
       A(I)=A(I+K) + D(I)
       B(I)=B(I) + A(I)
      END DO
```

For more information on optimization, see the *Fortran Programming Guide.*

The PREFETCH Directives

The `-xprefetch` option flag, page 114, enables a set of `PREFETCH` directives that advise the compiler to generate prefetch instructions for the specified data element. Prefetch instructions are only available on UltraSPARC platforms.

```
C$PRAGMA SPARC_PREFETCH_READ_ONCE(name)
C$PRAGMA SPARC_PREFETCH_READ_MANY(name)
C$PRAGMA SPARC_PREFETCH_WRITE_ONCE(name)
C$PRAGMA SPARC_PREFETCH_WRITE_MANY(name)
```

See also the *C User's Guide*, or the *SPARC Architecture Manual, Version 9* for further information about prefetch instructions.

Parallelization Directives

Parallelization directives explicitly request the compiler to attempt to parallelize the `DO` loop or the region of code that follows the directive. The syntax differs from general directives. Parallelization directives are only recognized when compilation options `-parallel` or `-explicitpar` are used. Details regarding Fortran parallelization can be found in the *Fortran Programming Guide*.

Note – *Fortran parallelization features require a Forte for High Performance Computing (HPC) license.*

The Fortran compilers support three styles of parallelization directives, Sun, Cray, and OpenMP.

Sun style parallelization directives are the default (explicitly selected with the compiler option `-mp=sun`). Sun directives have the directive *sentinel* `$PAR`.

Cray style parallelization directives, enabled by the `-mp=cray` compiler option, have the sentinel `MIC$`. Interpretations of similar directives differ between Sun and Cray styles. See the chapter on parallelization in the *Fortran Programming Guide* for details.

Fortran 95 also accepts OpenMP parallelization directives, described in the next section.

Sun/Cray parallelization directives have the following syntax:

- The first character must be in column one.
- The first character can be any one of `c`, `C`, `*`, or `!`.
- The next four characters may be either `$PAR` (Sun style), or `MIC$` (Cray style), without blanks, and in either upper or lower case.
- Next, the directive keyword and qualifiers, separated by blanks. The explicit parallelization directive keywords are:

  `TASKCOMMON`, `DOALL`, `DOSERIAL`, and `DOSERIAL*`

Each parallelization directive has its own set of optional qualifiers that follow the keyword.

Example: Specifying a loop with a shared variable:

```
C$PAR DOALL SHARED(yvalue)     Sun style
CMIC$ DOALL SHARED(yvalue)     Cray style
```

See Appendix E for a summary, and the *Fortran Programming Guide* for details about parallelization and these directives.

OpenMP Directives

The Sun WorkShop 6 Fortran 95 compiler recognizes the OpenMP Fortran shared memory multiprocessing API as specified by the OpenMP Architecture Review Board. See the OpenMP website for details: `http://www.openmp.org/`.

You must compile with the command-line option `-openmp`, to enable OpenMP directives. (`-openmp` is a macro flag that invokes the compiler options required by OpenMP; see page 79.)

A summary of OpenMP directives appears in Appendix E.

OpenMP directives can be used in conjunction with either Sun or Cray style parallelization directives, as long as these different directives are not nested within each other. To enable OpenMP with Sun or Cray directives, use `-mp=openmp,sun` or `-mp=openmp,cray` (no spaces), respectively. (See page 73)

`f95`: Library Interfaces and `system.inc`

The Fortran 95 compiler provides an include file, `system.inc`, that defines the interfaces for most non-intrinsic library routines. Declare this include file to insure that functions you call and their arguments are properly typed, especially when default data types are changed with `-xtypemap`.

For example, the following may produce an arithmetic exception because function `getpid()` is not explicitly typed:

```
        integer(4) mypid
        mypid = getpid()
        print *, mypid
```

The `getpid()` routine returns an integer value but the compiler assumes it returns a real value if no explicit type is declared for the function. This value is further converted to integer, most likely producing a floating-point error.

To correct this you should explicitly type `getuid()` and functions like it that you call:

```
        integer(4) mypid, getpid
        mypid = getpid()
        print *, mypid
```

Problems like these can be diagnosed with the `-Xlist` (global program checking) option. The Fortran 95 include file `'system.inc'` provides explicit interface definitions for these routines.

```
        include 'system.inc'
        integer(4) mypid
        mypid = getpid()
        print *, mypid
```

Including `system.inc` in program units calling routines in the Fortran library will automatically define the interfaces for you, and help the compiler diagnose type mismatches. (See the *Fortran Library Reference* for more information.)

Compiler Usage Tips

The next sections suggest a number of ways to use the Sun Fortran compilers efficiently. A complete compiler options reference follows in the next chapter.

Determining Hardware Platform

Some compiler flags allow the user to tune code generation to a specific set of hardware platform options. The utility command `fpversion` displays the hardware platform specifications for the native processor:

```
demo% fpversion
A SPARC-based CPU is available.
 CPU's clock rate appears to be approximately 467.1 MHz.
 Kernel says CPU's clock rate is 480.0 MHz.
 Kernel says main memory's clock rate is 120.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.
 FPU's frequency appears to be approximately 492.7 MHz.

 Use "-xtarget=ultra2i -xcache=16/32/1:2048/64/1" code-
generation option.

 Hostid = hardware_host_id.
```

The values printed depend on the load on the system at the moment `fpversion` is called.

See `fpversion`(1) and the *Numerical Computation Guide* for details.

Using Environment Variables

You can specify options by setting the `FFLAGS` or `OPTIONS` variables.

Either FFLAGS or OPTIONS can be used explicitly in the command line. When you are using the implicit compilation rules of make, FFLAGS is used automatically by the make program.

Example: Set FFLAGS: (C Shell)

```
demo% setenv FFLAGS '-fast -Xlist'
```

Example: Use FFLAGS explicitly:

```
demo% f95 $FFLAGS any.f
```

When using make, if the FFLAGS variable is set as above and the makefile's compilation rules are *implicit*, that is, there is no *explicit* compiler command line, then invoking make will result in a compilation equivalent to:

`f77 -fast -Xlist` *files...*

make is a very powerful program development tool that can easily be used with all Sun compilers. See the make(1) man page and the *Program Development* chapter in the *Fortran Programming Guide*.

Note – Default implicit rules assumed by make may not recognize files with extensions .f95 and .mod (Fortran 95 Module files). See the *Fortran Programming Guide* and the Fortran 95 readme file for details.

Memory Size

A compilation may need to use a lot of memory. This will depend on the optimization level chosen and the size and complexity of the files being compiled. On SPARC platforms, if the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent routines at the original level specified in the -O*n* option on the command line.

A workstation should have at least 24 megabytes of memory; 32 megabytes are recommended. Memory usage depends on the size of each procedure, the level of optimization, the limits set for virtual memory, the size of the disk swap file, and various other parameters.

Compiling a single source file containing many routines could cause the compiler to run out of memory or swap space.

If the compiler runs out of memory, try reducing the level of optimization, or split multiple-routine source files into files with one routine per file, using `fsplit`(1).

Swap Space Limits

The SunOS™ operating system command, `swap -s`, displays available swap space. See *swap*(1M).

Example: Use the `swap` command:

```
demo% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used,
1058708k available
```

To determine the actual real memory:

```
demo% /usr/sbin/dmesg | grep mem
mem = 655360K (0x28000000)
avail mem = 602476544
```

Increasing Swap Space

Use `mkfile`(1M) and `swap`(1M) to increase the size of the swap space on a workstation. You must become superuser to do this. `mkfile` creates a file of a specific size, and `swap -a` adds the file to the system swap space:

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

Control of Virtual Memory

Compiling very large routines (thousands of lines of code in a single procedure) at optimization level `-O3` or higher may require additional memory that could degrade compile-time performance. You can control this by limiting the amount of virtual memory available to a single process.

In a `sh` shell, use the `ulimit` command. See `sh`(1).

Example: Limit virtual memory to 16 Mbytes:

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

In a `csh` shell, use the `limit` command. See `csh`(1).

Example: Limit virtual memory to 16 Mbytes:

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

Each of these command lines causes the optimizer to try to recover at 16 Mbytes of data space.

This limit cannot be greater than the system's total available swap space and, in practice, must be small enough to permit normal use of the system while a large compilation is in progress. Be sure that no compilation consumes more than half the space.

Example: With 32 Mbytes of swap space, use the following commands:

In a `sh` shell:

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

In a `csh` shell:

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

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

In 64-bit Solaris environments, the soft limit for the size of an application data segment is 2 Gbytes. If your application needs to allocate more space, use the shell's `limit` or `ulimit` command to remove the limit.

For `csh` use:

```
demo% limit datasize unlimited
```

For `sh` or `ksh`, use:

```
demo$ ulimit -d unlimited
```

See the *Solaris 64-bit Developer's Guide* for more information.

CHAPTER 3

Fortran Compiler Options

This chapter details the command–line options for the `f77` and `f95` compilers.

- A description of the syntax used for compiler option flags starts on page 37
- Summaries of options arranged by functionality starts on page 39.
- The complete reference detailing each compiler option flag starts on page 45.

Some options are not available on both compilers (`f77` or `f95`). Check the reference section for availability.

Command Syntax

The general syntax of the compiler command line is:

```
f77 [options] list_of_files additional_options
f95 [options] list_of_files additional_options
```

Items in square brackets indicate optional parameters. The brackets are not part of the command. The *options* are a list of option keywords prefixed by dash (–). Some keyword options take the next item in the list as an argument. The *list_of_files* is a list of source, object, or library file names separated by blanks. Also, there are some options that must appear after the list of source files, and these could include additional lists of files (for example, `-B`, `-l`, and `-L`).

Options Syntax

Typical compiler option formats are:

TABLE 3-1 Options Syntax

Syntax Format	Example
–flag	`-g`
–flagvalue	`-Dnostep`
–flag=value	`-xunroll=4`
–flag value	`-o outfile`

The following typographical conventions are used when describing the individual options:

TABLE 3-2 Typographic Notations for Options

Notation	Meaning	Example: Text/Instance
[ ]	Square brackets contain arguments that are optional.	`-O[n]` `-O4, -O`
{ }	Curly brackets 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`

Brackets, pipe, and ellipsis are *meta characters* used in the descriptions of the options and are not part of the options themselves.

Some general guidelines for options are:

- `-l`*x* is the option to link with library `lib`*x*`.a`. It is always safer to put `-l`*x* after the list of file names to insure the order libraries are searched.

- In general, processing of the compiler options is from left to right, allowing selective overriding of macro options (options that include other options).
  - The above rule does not apply to linker options.
  - However, some options, `-I`, `-L`, and `-R` for example, accumulate values rather than override previous values when repeated on the same command line.

Source files, object files, and libraries are compiled and linked in the order in which they appear on the command line.

Options Summary

In this section, the compiler options are grouped by function to provide an easy reference. The details will be found on the pages in the following sections, as indicated.

The following table summarizes the `f77` and `f95` compiler options by functionality. The table does not include obsolete and legacy option flags. Some flags serve more than one purpose and appear more than once.

TABLE 3-3 Compiler Options Grouped by Functionality

Function	Option Flag
Compilation Mode:	
Compile only; do not produce an executable file	`-c`
Show commands built by the driver but do not compile	`-dryrun`
Specify name of object, library, or executable file to write	`-o` *filename*
Compile and generate only assembly code	`-S`
Strip symbol table from executable	`-s`
Suppress compiler messages, except error messages	`-silent`
Define path to directory for temporary files	`-temp=`*directory*
Show elapsed time for each compilation phase	`-time`
Show version number of compiler and its phases	`-V`
Verbose messages	`-v`
Compiled Code:	
Add/suppress trailing underscores on external names	`-ext_names=`*x*
Inline specified user functions	`-inline=`*list*

TABLE 3-3 Compiler Options Grouped by Functionality *(Continued)*

Function	Option Flag
Compile position independent code	`-KPIC/-kpic`
Inline certain math library routines	`-libmil`
STOP returns integer status value to shell	`-stop_status[=`*yn*`]`
Specify code address space	`-xcode=`*x*
Enable UltraSPARC prefetch instructions	`-xprefetch[=`*x*`]`
Specify use of optional registers	`-xregs=`*x*
Specify default data mappings	`-xtypemap=`*x*
Data Alignment:	
Specify alignment of data in COMMON blocks	`-aligncommon[=`*n*`]`
Force COMMON block data alignment to allow double word fetch/store	`-dalign`
Force alignment of all data on 8-byte boundaries	`-dbl_align_all`
Align COMMON block data on 8-byte boundaries	`-f`
Specify memory alignment and behavior	`-xmemalign[=`*ab*`]`
Debugging:	
Enable runtime subscript range checking	`-C`
Compile for debugging	`-g`
Compile for browsing with Sun WorkShop source browser	`-sb, -sbfast`
Flag use of undeclared variables	`-u`
Compile for Sun WorkShop Performance Analyzer	`-xF`
Generate source listings	`-Xlist`*x*
Enable debugging without object files	`-xs`
Diagnostics:	
Flag use of non-standard extensions	`-ansi`
Suppress specific error messages	`-erroff=`*list*
Display error tag names with error messages	`-errtags`
Show summary of compiler options	`-flags, -help`
Show version number of the compiler and its phases	`-V`
Verbose messages	`-v`
Verbose parallelization messages	`-vpara`

TABLE 3-3 Compiler Options Grouped by Functionality *(Continued)*

Function	Option Flag
Show/suppress warning messages	`-w`*n*
Enable runtime task common check	`-xcommonchk`
Display compiler README file	`-xhelp=readme`
Licensing:	
Show license server information	`-xlicinfo`
Linking and Libraries:	
Allow/require dynamic/static libraries	`-B`*x*
Allow only dynamic/static library linking	`-dy, -dn`
Build a dynamic (shared object) library	`-G`
Assign name to dynamic library	`-h`*name*
Add directory to library search path	`-L`*dir*
Link with library `lib`*name*`.a` or `lib`*name*`.so`	`-l`*name*
Build runtime library search path into executable	`-R`*dir*
Disable use of incremental linker, `ild`	`-xildoff`
Link with optimized math library	`-xlibmopt`
Link with Sun Performance Library	`-xlic_lib=sunperf`
Link editor option	`-z`*x*
Generate pure libraries with no relocations	`-ztext`
Numerics and Floating-Point:	
Use non-standard floating-point preferences	`-fnonstd`
Select SPARC non-standard floating point	`-fns`
Enable runtime floating-point overflow during input	`-fpover`
Select IEEE floating-point rounding mode	`-fpround=`*r*
Select floating-point optimization level	`-fsimple=`*n*
Select floating-point trapping mode	`-ftrap=`*t*
Promote single precision constants to double precision	`-r8const`
Enable interval arithmetic and set the appropriate floating-point environment (includes `-xinterval`)	`-xia[=`*e*`]`
Enable interval arithmetic extensions	`-xinterval[=`*e*`]`

TABLE 3-3 Compiler Options Grouped by Functionality *(Continued)*

Function	Option Flag
Optimization and Performance:	
Analyze loops for data dependencies	`-depend`
Optimize using a selection of options	`-fast`
Specify optimization level	`-O`*n*
Pad data layout for efficient use of cache	`-pad[=`*p*`]`
Allocate local variables on the memory stack	`-stackvar`
Enable loop unrolling	`-unroll[=`*m*`]`
Enable optimization across source files	`-xcrossfile[=`*n*`]`
Invoke interprocedural optimizations pass	`-xipo[=`*n*`]`
Set highest optimization level for `#pragma OPT`	`-xmaxopt[=`*n*`]`
Assert that no memory-based traps will occur	`-xsafe=mem`
Do no optimizations that increase code size	`-xspace`
Generate calls to vector library functions automatically	`-xvector[=`*yn*`]`
Parallelization: *(Note: Fortran parallelization features require a Forte for High Performance Computing license).*	
Enable automatic parallelization of DO loops	`-autopar`
Enable parallelization of loops explicitly marked with directives	`-explicitpar`
Show loop parallelization information	`-loopinfo`
Specify which style of directives to accept: Sun, Cray, OpenMP	`-mp=`*v*
Compile for hand-coded multithreaded programming	`-mt`
Accept OpenMP API directives and set appropriate environment (macro)	`-openmp`
Parallelize loops with `-autopar -explicitpar -depend` combination	`-parallel`
Recognize reduction operations in loops with automatic parallelization	`-reduction`
Verbose parallelization messages	`-vpara`
Source Code:	
Define preprocessor symbol	`-D`*name*`[=`*val*`]`
Undefine preprocessor symbol	`-U`*name*

TABLE 3-3 Compiler Options Grouped by Functionality *(Continued)*

Function	Option Flag
Accept extended (132 character) source lines	`-e`
Apply preprocessor to `.F` and/or `.F90` and `.F95` files but do not compile	`-F`
Accept fixed-format input (`f95`)	`-fixed`
Preprocess all source files with the `fpp` preprocessor	`-fpp`
Accept free-format input (`f95`)	`-free`
Add directory to include file search path	`-I`*dir*
Add directory to module search path	`-M`*dir*
Recognize upper and lower case as distinct	`-U`
Select preprocessor, `cpp` or `fpp`, to use	`-xpp[={fpp|cpp}]`
Allow recursive subprogram calls	`-xrecursive`
Target Platform:	
Optimize for the host system	`-native`
Specify target platform instruction set for the optimizer	`-xarch=`*a*
Specify target cache properties for optimizer	`-xcache=`*a*
Specify target processor for the optimizer	`-xchip=`*a*
Specify target platform for the optimizer	`-xtarget=`*a*

Commonly Used Options

The Sun Fortran compilers have many features that are selectable by optional command–line parameters. The short list below of commonly used options is a good place to start.

TABLE 3-4 Commonly Used Options

Action	Option
Debug—global program checking across routines for consistency of arguments, commons, and so on.	`–Xlist`
Debug—produce additional symbol table information to enable the `dbx` and Sun WorkShop debugging.	`–g`
Performance—invoke the optimizer to produce faster running programs.	`–O[`*n*`]`

TABLE 3-4 Commonly Used Options *(Continued)*

Action	Option
Performance—Produce efficient compilation and run times for the native platform, using a set of predetermined options.	`-fast`
Dynamic (`-Bdynamic`) or static (`-Bstatic`) library binding.	`-B`*x*
Compile only—Suppress linking; make a `.o` file for each source file.	`-c`
Output file—Name the executable output file *nm* instead of `a.out`.	`-o nm`
Source code—Compile fixed format Fortran 77 code with `f95`.	`-fixed`

Backward Compatibility and Legacy Options

The following options are provided for backward compatibility with earlier compiler releases, and certain Fortran legacy capabilities.

TABLE 3-5 Backward Compatibility Options

Action	Option
Double default data sizes: use `-xtypemap` instead.	`-r8` *or* `-dbl`
Allow assignment to constant arguments.	`-copyargs`
Treat hollerith constant as character or typeless in call argument lists.	`-xhasc[={yes|no}]`
External names—make external names without underscores.	`-ext_names=`*e*
Nonstandard arithmetic—allow nonstandard arithmetic.	`-fnonstd`
Optimize performance for the host system.	`-native`
Output—use old style list–directed output.	`-oldldo`
`DO` loops—use one trip `DO` loops.	`-onetrip`
Compile for SPARC V7 architecture	`-cg89`
Compile for SPARC V8 architecture	`-cg92`

Use of these option flags is not recommended and should be avoided.

Obsolescent Options

The following options are no longer supported by the `f77` and `f95` compilers. Their appearance on a compiler command does not cause an error, and no action is taken; they are ignored.

TABLE 3-6 Obsolescent Options

Original Intention	Option
Compile for analysis by looptool	`-Zlp`
Compile for Thread Analyzer	`–Ztha`
Disable exception traps (`f95`)	`–fnonstop`

Options Reference

This section shows all `f77` and `f95` compiler command–line option flags, including various risks, restrictions, caveats, interactions, examples, and other details. Each description indicates platform availability of the option.

The following table indicates availability of an option:

Legend	Option Availability
f77	only available with `f77`
f95	only available with `f95`
f77/f95	available with both `f77` and `f95`

Options that are not available for a compiler on a particular platform will still be accepted *silently* by the compiler. That is, the compiler will accept the option on the command–line on that platform without issuing a warning, but the option does nothing.

This options reference details each option flag.

-a

Profile by basic block using `tcov`, old style.

- **f77/f95**

This is the old style of basic block profiling for `tcov`. See `-xprofile=tcov` for information on the new style of profiling and the `tcov`(1) man page for more details. Also see the manual, *Analyzing Program Performance with Sun WorkShop*.

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

If set at compile–time, the `TCOVDIR` environment variable specifies the directory where the `.d` and `.tcov` files are located. If this variable is not set, then the `.d` files remain in the same directory as the `.f` files.

The `-xprofile=tcov` and the `-a` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov`, and others with `-a`. You cannot compile a single file with both options.

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

For details, see the chapter *Performance Profiling* in the *Fortran Programming Guide*.

-aligncommon[=*n*]

Specify the alignment of data in common blocks.

- **f77/f95**

n may be 1, 2, 4, 8, or 16, and indicates the maximum alignment (in bytes) for data elements within common blocks.

For example, `-aligncommon=4` would align all common block data elements with natural alignments of 4 bytes or more on 4-byte boundaries.

This option does not affect data with natural alignment smaller than the specified size.

Without `-aligncommon`, the compilers align common block data on (at most) 4-byte boundaries.

Specifying `-aligncommon` without a value defaults to 1 on all platforms: all common block data aligns on byte boundaries (no padding between elements).

`-aligncommon=16` reverts to `-aligncommon=8` on platforms that are not 64-bit enabled (platforms other than `v9`, `v9a`, or `v9b`).

-ansi

Identify many nonstandard extensions.

- **f77/f95**

Warning messages are issued for any uses of non–standard Fortran 77 or Fortran 95 extensions in the source code.

-arg=local

Preserve actual arguments over `ENTRY` statements.

- **f77**

When you compile a subprogram with alternate entry points with this option, `f77` uses *copy restore* to preserve the association of dummy and actual arguments. For example, the following program would require compilation with `-arg=local` to insure proper execution:

```
A = SETUP(ALPHA,BETA,GAMMA)
ZORK = FXGAMMA(GCONST)
...
FUNCTION SETUP(A1,A2,A3)
...
ENTRY FXGAMMA(F)
FXGAMMA = F*GAMMA
...
RETURN
END
```

Without this option, there is no guarantee that the correct values of the actual arguments from the `SETUP` call will be referenced when the routine is entered through `FXGAMMA`. Code that relies on `-arg=local` is nonstandard.

-autopar

Enable automatic loop parallelization. *Fortran parallelization features require a Forte for HPC license.*

- **f77/f95**

Finds and parallelizes appropriate loops for running in parallel on multiple processors. Analyzes loops for inter–iteration data dependencies and loop restructuring. If the optimization level is not specified `-O3` or higher, it will automatically be raised to `-O3`.

To improve performance, also specify the `–stackvar` option when using any of the parallelization options, including `–autopar`.

Avoid `-autopar` if the program already contains explicit calls to the `libthread` threads library. See note with `–mt` on page 74.

The `-autopar` option is not appropriate on a single–processor system, and the compiled code will generally run slower.

To run a parallelized program in a multithreaded environment, you must set the `PARALLEL` (or `OMP_NUM_THREADS`) environment variable prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set the `PARALLEL` or `OMP_NUM_THREADS` variable to the available number of processors on the target platform.

If you use `–autopar` and compile and link in *one* step, the multithreading library and the thread–safe Fortran runtime library will automatically be linked. If you use `-autopar` and compile and link in *separate* steps, then you must also link with `-autopar` to insure linking the appropriate libraries.

The `-reduction` option may also be useful with `–autopar`. Other parallelization options are `–parallel` and `–explicitpar`.

Refer to the *Fortran Programming Guide* for more information on parallelization.

-B{static|dynamic}

Prefer dynamic or require static library linking.

- **f77/f95**

No space is allowed between `–B` and `dynamic` or `static`. The default, without `–B` specified, is `–Bdynamic`.

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

Also note:

- If you specify `static`, but the linker finds only a dynamic library, then the library is not linked with a warning that the "library was not found."
- If you specify `dynamic`, but the linker finds only a static version, then that library is linked, with no warning.

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

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

These are loader and linker options. Compiling and linking in separate steps with `-B`*x* on the compile command will require it in the link step as well.

You cannot specify both `-Bdynamic` and `-dn` on the command line because `-dn` disables linking of dynamic libraries.

In a 64-bit Solaris environment, many system libraries are available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). This means that `–Bstatic` and `–dn` may cause linking errors in 64-bit Solaris environments. Applications must link with the dynamic libraries in these cases.

See the *Fortran Programming Guide* for more information on static and dynamic libraries.

–C

Check array references for out of range subscripts.

- **f77/f95**

Subscripting arrays beyond their declared sizes may result in unexpected results, including segmentation faults. The `–C` option checks for possible array subscript violations in the source code and during execution.

Specifying `–C` may make the executable file larger.

If the `–C` option is used, array subscript violations are treated as an error. If an array subscript range violation is detected in the source code during compilation, it is treated as a compilation error.

If an array subscript violation can only be determined at runtime, the compiler generates range–checking code into the executable program. This may cause an increase in execution time. As a result, it is appropriate to enable full array subscript checking while developing and debugging a program, then recompiling the final production executable without subscript checking.

-c

Compile only; produce object `.o` files, but suppress linking.

- **f77/f95**

  Suppress linking. Compile a `.o` file for each source file. If only a single source file is being compiled, the `–o` option can be used to specify the name of the `.o` file written.

-cg89

Compile for generic SPARC architecture. *(Obsolete)*

- **f77/f95**

  This option is a macro for: `–xarch=v7 –xchip=old –xcache=64/32/1` which is equivalent to `–xtarget=ss2` .

-cg92

Compile for SPARC V8 architecture. *(Obsolete)*

- **f77/f95**

  This option is a macro for:
  `–xarch=v8 –xchip=super –xcache=16/32/4:1024/32/1` which is equivalent to `–xtarget=ss1000` .

-copyargs

Allow assignment to constant arguments.

- **f77/f95**

  Allow a subprogram to change a dummy argument that is a constant. This option is provided only to allow legacy code to compile and execute without a runtime error.

  - Without `–copyargs`, if you pass a constant argument to a subroutine, and then within the subroutine try to change that constant, the run aborts.
  - With `–copyargs`, if you pass a constant argument to a subroutine, and then within the subroutine change that constant, the run does not necessarily abort.

  Code that aborts unless compiled with `–copyargs` is, of course, not Fortran standard compliant. Also, such code is often unpredictable.

-D*name*[**=***def*]

Define symbol *name* for the preprocessor.

- **f77/f95**

This option only applies to `.F`, `.F90`, and `.F95` source files.

`-D`*name*`=`*def* Define *name* to have value *def*

`-D`*name* Define *name* to be `1`

On the command line, this option will define *name* as if:

`#define` *name*`[=`*def*`]`

had appears in the source file. If no `=`*def* specified, the name *name* is defined as the value 1. The macro symbol *name* is passed on to the preprocessor `fpp` (or `cpp` — see the `-xpp` option) for expansion.

The predefined macro symbols have two leading underscores. The Fortran syntax may not support the actual values of these macros—they should appear only in `fpp` or `cpp` preprocessor directives.

- The product version is predefined (in hex) in `__SUNPRO_F77`, `__SUNPRO_F90`, and `__SUNPRO_F95`. For example `__SUNPRO_F77` is `0x600` for the Sun WorkShop 6 release.
- The following macros are predefined on appropriate systems:

  `__sparc`, `__unix`, `__sun`, `__SVR4`,
  `__SunOS_5_6`, `__SunOS_5_7`, `__SunOS_5_8`

  For instance, the value `__sparc` is defined on SPARC systems. You can use these values in such preprocessor conditionals as the following:

  ```
  #ifdef __sparc
  ```

- The following are predefined with no underscores, but they may be deleted in a future release: `sparc`, `unix`, `sun`
- On SPARC V9 systems, the `__sparcv9` macro is also defined.

The compilers use the `fpp`(1) preprocessor by default. Like the C preprocessor `cpp`(1), `fpp` expands source code macros and enables conditional compilation of code. Unlike `cpp`, `fpp` understands Fortran syntax, and is preferred as a Fortran preprocessor. Use the –`xpp=cpp` flag to force the compiler to specifically use `cpp` rather than `fpp`.

-dalign

Align COMMON block data and generate faster multi-word load/stores.

- **f77/f95**

This flag changes the data layout in COMMON blocks (and EQUIVALENCE classes), and enables the compiler to generate faster multi-word load/stores for that data.

The data layout effect is that of the `-f` flag: double- and quad-precision data in COMMON blocks and EQUIVALENCE classes are laid out in memory along their "natural" alignment, which is on 8-byte boundaries (or on 16-byte boundaries for quad-precision when compiling for 64-bit environments with `-xarch=v9` or `v9a`). The default alignment of data in COMMON blocks is on 4-byte boundaries. The compiler is also allowed to assume natural alignment and generate faster multi-word load/stores to reference the data.

Note – `-dalign` may result in nonstandard alignment of data, which could cause problems with variables in `EQUIVALENCE` or `COMMON` and may render the program non-portable if `–dalign` is required.

`-dalign` is a macro equivalent to: `-xmemalign=8s -aligncommon=16.` See `-aligncommon`, page 46, and `-xmemalign`, page 111.

Using both `–dbl` (`f77`) and `–dalign` also causes default `INTEGER` variables to be 8-byte aligned and 64-bits. `-xtypemap` is preferred over `-dbl`:

```
-xtypemap=real:64,double:128,integer:64
```

If you compile one subprogram with `–dalign`, compile all subprograms of the program with `–dalign`. This option is included in the `–fast` option.

-db

Generate optional CIF file.

- **f95**

Generates an optional compiler information file (CIF) with the extension `.T`. This file is sometimes needed by the Sun WorkShop Source Browser.

-dbl

Double the default size for REAL, INTEGER, DOUBLE, and COMPLEX.

- f77

Note – This option, and –r8, is now considered obsolete and may be removed in future releases. Use the more general –xtypemap option instead.

-dbl promotes the default byte size for REAL, INTEGER, DOUBLE, and COMPLEX variables declared *without an explicit byte size* as follows:

TABLE 3-7 Default Data Sizes and –dbl (Bytes)

Without –dbl option		With –dbl option
Data Type	default	SPARC
INTEGER	4	8
REAL	4	8
DOUBLE	8	16

This option applies to variables, parameters, constants, and functions.

Also, LOGICAL is treated as INTEGER, COMPLEX as two REALs, and DOUBLE COMPLEX as two DOUBLEs.

Compare -dbl with –r8: -dbl and –r8 can be expressed in terms of the more general –xtypemap= option:

- -dbl *same as:* -xtypemap=real:64,double:128,integer:64
- -r8 *same as:* -xtypemap=real:64,double:128,integer:mixed

These options promote default DOUBLE PRECISION data to QUAD PRECISION (128 bits). This may be unwanted and may cause performance degradation. It might be more appropriate to use -xtypemap=real:64,double:64,integer:64 instead of -dbl in these cases.

For all of the floating point data types, -dbl works the same as -r8; using both -r8 and -dbl produces the same results as using only -dbl.

- For INTEGER and LOGICAL data types, -dbl is different from -r8:
  - -dbl allocates 8 bytes, and does 8–byte arithmetic
  - -r8 allocates 8 bytes, and does only 4–byte arithmetic (“mixed”)

In general, if you compile *one* subprogram with `-dbl`, then be sure to compile *all* subprograms of that program with `-dbl`. This is particularly important with programs communicating through files with unformatted I/O — if one program is compiled with `-dbl`, then the other program must similarly be compiled. Be also aware that this option alters the default data size of function names, including calls to library functions, unless the function name is typed explicitly with a data size.

`-dbl_align_all={yes|no}`

Force alignment of data on 8–byte boundaries

- **f77/f95**

The value is either `yes` or `no`. If `yes`, all variables will be aligned on 8–byte boundaries. Default is `-dbl_align_all=no`.

When compiling for 64-bit environments with `-xarch=v9` or `v9a`, this flag will align quad-precision data on 16-byte boundaries.

This flag does not alter the layout of data in COMMON blocks or user-defined structures.

On SPARC, use with `-dalign` to enable added efficiency with multi-word load/stores.

If used, all routines must be compiled with this flag.

`-depend`

Analyze loops for data dependencies.

- **f77/f95**

Analyze loops for data dependencies and do loop restructuring. This option will raise the optimization level to `O3` if no optimization level is specified, or if it is specified less than `O3`. `-depend` is also included with `-fast`, `-autopar` and `-parallel`. (See the *Fortran Programming Guide*.)

`-dn`

Disallow dynamic libraries. See `-d{y|n}`, page 55.

- **f77/f95**

-dryrun

Show commands built by `f77` or `f95` command-line driver, but do not compile.

- **f77/f95**

Useful when debugging, this option displays the commands and suboptions the compiler will invoke to perform the compilation.

-d{y|n}

Allow or disallow *dynamic* libraries for the entire executable

- **f77/f95**
  - `-dy`: Yes, *allow* dynamic/shared libraries.
  - `-dn`: No, *do not allow* dynamic/shared libraries.

The default, if not specified, is `-dy`.

Unlike –B*x*, this option applies to the *whole* executable and need appear only once on the command line.

`-dy|-dn` are loader and linker options. If you compile and link in separate steps with these options, then you need the same option in the link step.

In a 64-bit Solaris environment, many system libraries are not available only as shared dynamic libraries. These include `libm.so` and `libc.so` (`libm.a` and `libc.a` are not provided). This means that `-dn` and `-Bstatic` may cause linking errors in 64-bit Solaris environments. Applications must link with the dynamic libraries in these cases.

-e

Accept extended length input source line.

- **f77/f95**

Accept source lines up to 132 characters long. The compiler pads on the right with trailing blanks to column 132. If you use continuation lines while compiling with `-e`, then do not split character constants across lines, otherwise, unnecessary blanks may be inserted in the constants.

-erroff=*taglist*

Suppress warning messages listed by tag name.

- **f77/f95**

Suppress the display of warning messages specified in the comma–separated list of tag names *taglist*. If *taglist* consists of `%none`, no warnings are suppressed. If *taglist* consists of `%all`, all warnings are suppressed (this is equivalent to the `-w` option.)

Example:

```
f77 -erroff=WDECL_LOCAL_NOTUSED ink.f
```

Use the `-errtags` option to see the tag names associated with warning messages.

-errtags[={yes|no}]

Display the message tag with each warning message.

- **f77/f95**

With `-errtags=yes`, the compiler's internal error tag name will appear along with warning messages. The default is not to display the tag (`-errtags=no`).

```
demo% f77 -errtags ink.f
ink.f:
 MAIN:
"ink.f", line 11: Warning: local variable "i" never used
(WDECL_LOCAL_NOTUSED)  <– The warning message's tag name
```

`-errtags` alone stands for `-errtags=yes`.

-explicitpar

Parallelize loops or regions explicitly marked by Sun, Cray, and/or OpenMP directives. *Fortran parallelization features require a Forte for HPC license.*

- **f77/f95**

The compiler will generate parallel code even if there are data dependencies in the DO loop that would cause the loop to generate incorrect results when run in parallel. With explicit parallelization, it is the user's responsibility to correctly analyze loops for data dependency problems before marking them with parallelization directives.

Parallelization is appropriate only on multiprocessor systems.

This option enables Sun, Cray, and/or OpenMP explicit parallelization directives. DO loops immediately preceded by parallelization directives will have threaded code generated for them.

Note: This option should not be used to compile programs that already do their own multithreading with calls to the `libthread` library.

To run a parallelized program in a multithreaded environment, you must set the `PARALLEL` (or `OMP_NUM_THREADS`) environment variable prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set the `PARALLEL` or `OMP_NUM_THREADS` variable to the available number of processors on the target platform.

If you use `-explicitpar` and compile and link in *one* step, then linking automatically includes the multithreading library and the thread–safe Fortran runtime library. If you use `-explicitpar` and compile and link in *separate* steps, then you must also *link* with `-explicitpar`.

To improve performance, also specify the `–stackvar` option when using any of the parallelization options, including `–explicitpar`.

Use the `-mp` option (page 73) to select the style of parallelization directives enabled: Sun, Cray, or OpenMP.

If the optimization level is not `-O3` or higher, it is raised to `-O3` automatically.

For details, see the *Parallelization* chapter in the *Fortran Programming Guide*.

`-ext_names=`*e*

Create external names with or without trailing underscores.

- **f77/f95**

e must be either `plain` or `underscores`. The default is `underscores`.

`-ext_names=plain`: Do not add trailing underscore.

`-ext_names=underscores`: Add trailing underscore.

An external name is a name of a subroutine, function, block data subprogram, or labeled common. This option affects both the name of the routine's entry point and the name used in calls to it. This option may be used to allow Fortran 77 routines to call and be called by other language routines.

-F

Invoke the source file preprocessor, but do not compile.

- **f77/f95**

Apply the `fpp` preprocessor to `.F` files (and `.f95` files with `f95`) and write the processed result on a file with the same name but with suffix changed to `.f` (or `.f95`), but do not compile.

Example:

```
f77 -F source.F
```

writes the processed source file to `source.f`

`fpp` is the default preprocessor for Fortran. The C preprocessor, `cpp`, can be selected instead by specifying `–xpp=cpp`.

-f

Align data in COMMON blocks.

- **f77/f95**

Align double- and quad-precision data in COMMON blocks.

This flag changes the data layout in COMMON blocks and EQUIVALENCE classes: double- and quad-precision data in COMMON blocks and EQUIVALENCE classes are laid out in memory along their "natural" alignment, which is on 8-byte boundaries (or on 16-byte boundaries for quad-precision when compiling for 64-bit environments with `-xarch=v9` or `v9a`). The default alignment of data in COMMON blocks is on 4-byte boundaries.

`-f` is equivalent to `-aligncommon=16.`

Note – `-f` may result in nonstandard alignment of data, which could cause problems with variables in `EQUIVALENCE` or `COMMON` and may render the program non-portable if `–f` is required.

Using `–dbl` (`f77`) with `–f` aligns all 64–bit integer data on 8–byte boundaries as well.

Compiling *any* part of a program with `-f` requires compiling *all* subprograms of that program with `-f`.

By itself, this option does not enable the compiler to generate faster multi-word fetch/store instructions on double and quad precision data. The `-dalign` option does this and invokes `-f` as well. Use of `-dalign` is preferred over the older `-f`. See `-dalign`, page 52. Because `-dalign` is part of the `-fast` option, so is `-f`.

`-fast`

Select options that optimize execution performance.

- **f77/f95**

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 `-v` (verbose) 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=%none` (`f77`) or `-ftrap=common` (`f95`)
- `–libmil`
- `–xtarget=native`
- `–O5`
- `–xlibmopt`
- `-pad=local`
- `-xvector=yes`
- `-xprefetch=yes`

Details about the options selected by `-fast`:

- The `-xtarget=native` hardware target.
  If the program is intended to run on a different target than the compilation machine, follow the `-fast` with a code–generator option. For example:
  ```
  f77 -fast -xtarget=ultra ...
  ```
- The `-O5` optimization level option. *(This is a change from previous compiler releases that set* `-O3` *or* `-O4` *with* `-fast`*.)*
- The `-depend` option analyzes loops for data dependencies and possible restructuring.
- The `-libmil` option for system–supplied inline expansion templates.
  For C functions that depend on exception handling, follow `-fast` by `-nolibmil` (as in `-fast -nolibmil`). With `-libmil`, exceptions cannot be detected with `errno` or `matherr`(3m).
- The `-fsimple=2` option for aggressive floating–point optimizations. `-fsimple=2` is unsuitable if strict IEEE 754 standards compliance is required. See page 64. *(This is a change from previous releases that set* `-fsimple=1` *with* `-fast`*.)*
- The `-dalign` option to generate double loads and stores for double and quad data in common blocks. Using this option can generate nonstandard Fortran data alignment in common blocks.
- The `-xlibmopt` option selects optimized math library routines.
- `-pad=local` inserts padding between local variables, where appropriate, to improve cache usage. *(This was not set by* `-fast` *in previous releases.)*
- `-xvector=yes` transforms certain math library calls within DO loops to single calls to a vectorized library equivalent routine with vector arguments. *(This was not set by* `-fast` *in previous releases.)*
- `-fns` selects non-standard SPARC floating-point arithmetic exception handling and gradual underflow. See page 62.
- `-ftrap=%none` to turn off all trapping for Fortran 77. Trapping on common floating-point exceptions, `-ftrap=common`, is the used with Fortran 95.
- `-xprefetch=yes` enables the compiler to generate hardware prefetch instructions where appropriate.

It is possible to add or subtract from this list by following the `-fast` option with other options, as in:

```
f95 -fast -fsimple=1 -xnolibmopt ...
```

which overrides the `-fsimple=2` option and disables the `-xlibmopt` selected by `-fast`.

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.

-fixed

Specify fixed–format Fortran 95 source input files.

- **f95**

  All source files on the command–line will be interpreted as `f77` fixed format regardless of filename extension. Normally, `f95` interprets only `.f` files as fixed format, `.f95` as free format.

-flags

Synonym for `-help`.

- **f77/f95**

-fnonstd

Initialize floating–point hardware to non–standard preferences.

- **f77/f95**

  This option is a synonym for the combination of the following option flags:

  - `-fns -ftrap=common`

  Specifying `-fnonstd` is approximately equivalent to the following two calls at the beginning of a Fortran main program.

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

  The `nonstandard_arithmetic()` routine replaces the obsolete `abrupt_underflow()` routine of earlier releases.

  To be effective, the main program must be compiled with this option.

  Using this option initializes the floating-point hardware to:

  - Abort (trap) on floating-point exceptions.
  - Flush underflow results to zero if it will improve speed, rather than produce a subnormal number as the IEEE standard requires.

  See `-fns` for more information about gradual underflow and subnormal numbers.

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

For more information, see the *ieee_handler*(3m) and *ieee_functions*(3m) man pages, the *Numerical Computation Guide*, and the *Fortran Programming Guide*.

-fns[={no|yes}]

Select the SPARC nonstandard floating–point mode.

- **f77/f95**

The default is the SPARC standard floating–point mode (`-fns=no`). (See the *Floating–Point Arithmetic* chapter of the *Fortran Programming Guide*.)

Optional use of `=yes` or `=no` provides a way of toggling the `-fns` flag following some other macro flag that includes it, such as `-fast`. `-fns` is the same as `-fns=yes`.

This option flag enables nonstandard floating-point mode when the program begins execution. On some SPARC systems, specifying nonstandard floating-point mode disables "gradual underflow", causing tiny results to be flushed to zero rather than producing subnormal numbers. It also causes subnormal operands to be silently replaced by zero. On those SPARC systems that do not support gradual underflow and subnormal numbers in hardware, use of this option can significantly improve the performance of some programs.

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

TABLE 3-8 Subnormal REAL and DOUBLE

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

See the *Numerical Computation Guide* for details on subnormal numbers, and the *Fortran Programming Guide* chapter *Floating–Point Arithmetic* for more information about this and similar options. (Some arithmeticians use the term *denormalized number* for *subnormal number*.)

The standard initialization of floating–point preferences is the default:

- IEEE 754 floating–point arithmetic is *nonstop* (do not abort on exception).
- Underflows are gradual.

To be effective, the main program must be compiled with this option.

-fpover[={yes|no}]

Detect floating-point overflow in formatted input.

- **f77/f95**

With `-fpover=yes` specified, the I/O library will detect runtime floating-point overflows in formatted input and return an error condition (1031). The default is no such overflow detection (`-fpover=no`). `-fpover` is equivalent to `-fpover=yes.`

-fpp

Force preprocessing of input with `fpp`.

- **f95**

Pass all the input source files listed on the f95 command line through the `fpp` preprocessor, regardless of file extension. (Normally, only files with `.F`, `.F90`, or `.F95` extension are automatically preprocessed by `fpp`.) See also `-xpp`, page 113.

-free

Specify free–format source input files.

- **f95**

All source files on the command–line will be interpreted as `f95` free format regardless of filename extension. Normally, `f95` interprets `.f` files as fixed format, `.f95` as free format.

-fround=*r*

Set the IEEE rounding mode in effect at startup.

- **f77/f95**

r must be one of: `nearest`, `tozero`, `negative`, `positive`.

The default is `-fround=nearest`.

To be effective, compile the main program with this option.

This option sets the IEEE 754 rounding mode that:

- Can be used by the compiler in evaluating constant expressions.
- Is established at runtime during the program initialization.

When *r* is `tozero`, `negative`, or `positive`, the option sets the rounding direction to *round-to-zero*, *round-to-negative-infinity*, or *round-to-positive-infinity*, respectively, when the program begins execution. When `-fround` is not specified,

`-fround=nearest` is used as the default and the rounding direction is *round-to-nearest*. The meanings are the same as those for the `ieee_flags` function. (See the *Floating–Point Arithmetic* chapter of the *Fortran Programming Guide*.)

`-fsimple[=`*n*`]`

Select floating–point optimization preferences.

- **f77/f95**

Allow the optimizer to make simplifying assumptions concerning floating–point arithmetic. (See the *Floating–Point Arithmetic* chapter of the *Fortran Programming Guide*.)

For consistent results, compile all units of a program with the same `–fsimple` option.

If *n* is present, it must be 0, 1, or 2. The defaults are:

- Without the `–fsimple` flag, the compiler defaults to `–fsimple=0`
- With `–fsimple` alone, the compiler defaults to `–fsimple=1`

The different floating–point simplification levels are:

`–fsimple=0`

Permit no simplifying assumptions. Preserve strict IEEE 754 conformance.

`–fsimple=1`

Allow conservative simplifications. The resulting code does not strictly conform to IEEE 754, but numeric results of most programs are unchanged.

With `–fsimple=1`, the optimizer can assume the following:

- IEEE 754 default rounding/trapping modes do not change after process initialization.
- Computations producing no visible result other than potential floating point exceptions may be deleted.
- Computations with Infinity or NaNs ("Not a Number") as operands need not propagate NaNs to their results; e.g., `x*0` may be replaced by `0`.
- Computations do not depend on sign of zero.

With `–fsimple=1`, the optimizer is *not* allowed to optimize completely without regard to roundoff or exceptions. In particular, a floating–point computation cannot be replaced by one that produces different results with rounding modes held constant at run time.

`-fsimple=2`

Permit aggressive floating point optimizations. This can cause some programs to produce different numeric results due to changes in the way expressions are evaluated. In particular, the Fortran standard rule requiring compilers to honor explicit parentheses around subexpressions to control expression evaluation order may be broken with `-fsimple=2`. This could result in numerical rounding differences with programs that depend on this rule.

For example, with `-fsimple=2`, the compiler may evaluate `C-(A-B)` as `(C-A)+B`, breaking the standard's rule about explicit parentheses, if the resulting code is better optimized. The compiler might also replace repeated computations of *x/y* with *x*z*, where *z=1/y* is computed once and saved in a temporary, to eliminate the costly divide operations.

Programs that depend on particular properties of floating-point arithmetic should not be compiled with `-fsimple=2`.

Even with `-fsimple=2`, the optimizer still is not permitted to introduce a floating point exception in a program that otherwise produces none.

`-fast` sets `-fsimple=2`.

`-ftrap=`*t*

Set floating–point trapping mode in effect at startup.

- **f77/f95**

t is a comma–separated list that consists of one or more of the following:

`%all`, `%none`, `common`, [`no%`]`invalid`, [`no%`]`overflow`, [`no%`]`underflow`, [`no%`]`division`, [`no%`]`inexact`.

`-ftrap=common` is a macro for `-ftrap=invalid,overflow,underflow,division`.

Where the `%` is shown, it is a required character.

The `f77` default is `-ftrap=%none`. The `f95` default is `-ftrap=common`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left–to–right. The common exceptions, by definition, are invalid, division by zero, and overflow. For example: `-ftrap=overflow`.

Example: `-ftrap=%all,no%inexact` means set all traps, except `inexact`.

The meanings for `-ftrap=`*t* are the same as for `ieee_flags()`, except that:

- `%all` turns on all the trapping modes, and will cause trapping of spurious and expected exceptions. Use `common` instead.

- `%none`, the `f77` default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

To be effective, compile the main program with this option.

For further information, see the *Floating–Point Arithmetic* chapter in the *Fortran Programming Guide*.

`-G`

Build a dynamic shared library instead of an executable file.

- **f77/f95**

Direct the linker to build a *shared dynamic* library. Without `-G`, the linker builds an executable file. With `-G`, it builds a dynamic library. Use `-o` with `-G` to specify the name of the file to be written. See the *Fortran Programming Guide* chapter *Libraries* for details.

`-g`

Compile for debugging and performance analysis.

- **f77/f95**

Produce additional symbol table information for debugging with `dbx`(1) or the Sun WorkShop debugging utility and for performance analysis with the Sun WorkShop Performance Analyzer.

Although some debugging is possible without specifying `-g`, the full capabilities of `dbx` and `debugger` are only available to those compilation units compiled with `-g`.

Some capabilities of other options specified along with `-g` may be limited. See the `dbx` documentation for details.

The `-g` option makes `-xildon` the default incremental linker option when .o object files appear on the command line (see page 106). That is, with `-g`, the compiler default behavior is to automatically invoke `ild` in place of `ld`, unless the `-G` option is present, or any source file is named on the command line.

To use the full capabilities of the Forte Developer 6 (Sun WorkShop 6) Performance Analyzer, compile with `-g`. While some performance analysis features do not require `-g`, you must compile with `-g` to view annotated source, some function level information, and compiler commentary messages. (See the `analyzer`(1) man page and *Analyzing Program Performance With Sun WorkShop*.)

The commentary messages generated with `-g` describe the optimizations and transformations the compiler made while compiling your program. The messages, interleaved with the source code, can be displayed by the `er_src`(1) command.

Note that commentary messages only appear if the compiler actually performed any optimizations. You are more likely to see commentary messages when you request high optimization levels, such as with `-xO4`, or `-fast`.

`-h`*name*

Specify the name of the generated dynamic shared library.

- **f77/f95**

This option is passed on to the linker. For details, see the Solaris *Linker and Libraries Guide*, and the *Fortran Programming Guide* chapter *Libraries*.

The `-h`*name* option records the name *name* to the shared dynamic library being created as the internal name of the library. A space between `-h` and *name* is optional (except if the library name is `elp`, for which the space will be needed). In general, *name* must be the same as what follows the `-o`. Use of this option is meaningless without also specifying `-G`.

Without the `-h`*name* option, no internal name is recorded in the library file.

If the library has an internal name, whenever an executable program referencing the library is run the runtime linker will search for a library with the same internal name in any path the linker is searching. With an internal name specified, searching for the library at runtime linking is more flexible. This option can also be used to specify *versions* of shared libraries.

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

`-help`

Display a summary list of compiler options.

- **f77/f95**

Displays a list of option summaries. See also `-xhelp=`*h* on page 105.

`-I`*dir*

Add *dir* to the INCLUDE file search path.

- **f77/f95**

Insert the directory *dir* at the start of the `INCLUDE` file search path. No space is allowed between `-I` and *dir*. Invalid directories are ignored with no warning message.

The *include file search path* is the list of directories searched for `INCLUDE` files—file names appearing on preprocessor `#include` directives, or Fortran `INCLUDE` statements.

Example: Search for `INCLUDE` files in `/usr/app/include`:

```
demo% f95 -I/usr/app/include growth.F
```

Multiple `-I`*dir* options may appear on the command line. Each adds to the top of the search path list (first path searched).

The search order for relative path on `INCLUDE` or #include is:

1. The directory that contains the source file
2. The directories that are named in the `-I`*dir* options
3. The directories in the default list

The default list for `-I`*dir* depends on the installation directory for the compiler. In a standard install, compiler software packages reside in the `/opt` directory; however, systems administrators may decide to install packages in other locations. The default search paths for `INCLUDE` files used by the compilers are:

- *for* `f77`: <*install_dir*>**`/SUNWspro/`**<*release*>**`/include/f77  /usr/include`**
- *for* `f95`: <*install_dir*>**`/SUNWspro/`**<*release*>**`/include/f90  /usr/include`**

where <*install_dir*> is the path to the installed packages (typically `/opt` in a normal install), and <*release*> is a path that varies with each software release.

`-i2`

Set the default integer size to two bytes.

- **f77**

Set the default size to 2 bytes for integer and logical constants and variables declared without an explicit size. (`INTEGER*`*n* `Y` still declares Y to be *n* bytes regardless of the `-i2`.) This option may degrade performance. It is generally recommended to declare specific variables `INTEGER*2` rather than use `-i2`.

-i4

Set the default integer size to four bytes.

- **f77**

Set the default size to 4 bytes for integer and logical constants and variables declared without an explicit size. (`INTEGER*`*n* `Y` still declares Y to be *n* bytes regardless of the `-i4`.).

Although 4 bytes *is* the default size for `INTEGER` and `LOGICAL`, this option can be used for overriding settings made by options like `-dbl` and `-r8`, which set these defaults to 8:

```
demo% f77 -dbl -i4 *.f
Command line warning: -i4 overrides integer part of -dbl
...
```

-inline=[%auto][[,][no%]*f1,...*[no%]*fn*]

Enable or disable inlining of specified routines.

- **f77/f95**

Request the optimizer to inline the user–written routines named in the *f1,...,fn* list. Prefixing a routine name with `no%` disables inlining of that routine.

Inlining is an optimization technique whereby the compiler effectively replaces a subprogram reference such as a `CALL` or function call with the actual subprogram code itself. Inlining often provides the optimizer more opportunities to produce efficient code.

The lists are a comma–separated list of functions and subroutines. To inhibit inlining of a function, prefix its name with `no%`.

Example: Inline the routines `xbar, zbar, vpoint`:

```
demo% f95 -O3 -inline=xbar,zbar,vpoint *.f
```

Following are the restrictions; no warnings are issued:

- Optimization must be `-O3` or greater.
- The source for the routine must be in the file being compiled, unless `-xcrossfile` is also specified.
- The compiler determines if actual inlining is profitable and safe.

The appearance of `-inline` with `-O4` disables the automatic inlining that the compiler would normally perform, unless `%auto` is also specified . With `-O4`, the compilers normally try to inline all appropriate user–written subroutines and functions. Adding `–inline` with `–O4` may degrade performance by restricting the optimizer's inlining to only those routines in the list. In this case, use the `%auto` suboption to enable automatic inlining at `-O4` and `-O5`.

```
demo% f95 -O4 -inline=%auto,no%zpoint *.f
```

In the example above, the user has enabled `-O4`'s automatic inlining while disabling any possible inlining of the routine `zpoint()` that the compiler might attempt.

`-Kpic`

Synonym for `–pic`.

- **f77/f95**

`-KPIC`

Synonym for `–PIC`.

- **f77/f95**

`-L`*dir*

Add *dir* to list of directories to search for libraries.

- **f77/f95**

Adds *dir* to the *front* of the list of object–library search directories. A space between `–L` and *dir* is optional. This option is passed to the linker. See also `–l`*x* on page 71.

While building the executable file, *ld*(1) searches *dir* for archive libraries (`.a` files) and shared libraries (`.so` files). `ld` searches *dir* before searching the default directories. (See the *Fortran Programming Guide* chapter *Libraries* for information on library search order.) For the relative order between `LD_LIBRARY_PATH` and `–L`*dir*, see *ld*(1).

Note – Specifying `/usr/lib` or `/usr/ccs/lib` with `–L`*dir* may prevent linking the unbundled `libm`. These directories are searched by default.

Example: Use `-L`*dir* to specify library search directories:

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

-l*x*

Add library `lib`*x*`.a` to linker's list of search libraries.

- **f77/f95**

Pass `-l`*x* to the linker to specify additional libraries for `ld` to search for unresolved references. *ld* links with object library `lib`*x*. If shared library `lib`*x*`.so` is available (and `–Bstatic` or `–dn` are not specified), `ld` uses it, otherwise, `ld` uses static library `lib`*x*`.a`. If it uses a shared library, the name is built in to `a.out`. No space is allowed between `-l` and *x* character strings.

Example: Link with the library `libV77`:

```
demo% f77 any.f –lV77
```

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

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

```
demo% f77 any.f –ly –lz
```

See also the *Libraries* chapter in the *Fortran Programming Guide* for information on library search paths and search order.

-libmil

Inline selected `libm` library routines for optimization.

- **f77/f95**

There are inline templates for some of the `libm` library routines. This option selects those inline templates that produce the fastest executable for the floating–point options and platform currently being used.

For more information, see the man pages `libm_single`(3F) and `libm_double`(3F)

-loopinfo

Show parallelization results. *The parallelization features of the Fortran compilers require a Forte for HPC license.*

- **f77/f95**

Show which loops were and were not parallelized with the `-parallel`, `-autopar`, or `-explicitpar` options. (Option `-loopinfo` must appear with one of these parallelization options.)

`-loopinfo` displays a list of messages on standard error:

```
demo% f95 -o shalow -fast -parallel -loopinfo shalow.f
...
"shalow.f", line 325: not parallelized, not profitable (inlined loop)
"shalow.f", line 172: PARALLELIZED, and serial version generated
"shalow.f", line 173: not parallelized, not profitable
"shalow.f", line 181: PARALLELIZED, fused
"shalow.f", line 182: not parallelized, not profitable
"shalow.f", line 193: not parallelized, not profitable
"shalow.f", line 199: PARALLELIZED, and serial version generated
"shalow.f", line 200: not parallelized, not profitable
"shalow.f", line 226: PARALLELIZED, and serial version generated
"shalow.f", line 227: not parallelized, not profitable
...etc
```

Use the *error*(1) utility with `f77` compilations to merge this list with the source file to produce an annotated source listing with each loop tagged as parallelized or not.

Example: Passing standard error to the `error` utility:

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

Be aware that `error` rewrites the input source file. For details on `error`, see the `error`(1) man page and the *Fortran Programming Guide* chapter on debugging.

-M*dir*

Add *dir* to directories searched for Fortran 95 modules.

- **f95**

Add *dir* to the list of directories to be searched for module files. No space appears between the –`M` and *dir*.

The directories listed with `-M` are searched after the current directory. Compiling a source file containing a module generates a `.mod` module file for each `MODULE` encountered. See in Appendix C, "Module Files" on page 167 for more information about modules in Fortran 95.

`-misalign`

Allow misaligned data.

- **f77**

The `-misalign` option permits misaligned data in memory that would otherwise produce an error. Particular uses of `COMMON` and `EQUIVALENCE` statements may cause data to be misaligned (with a compiler diagnostic). With `-misalign`, the compiler will allow intentional misalignment and will not add padding in COMMON blocks to insure proper data alignment. However, this seriously degrades performance; recoding to eliminate the cause of data misalignment is a better alternative.

If used, all routines in a program must be compiled with this option. If you compile and link in separate steps, compiling with the `-misalign` option requires the option on the link step as well.

`-misalign` is a macro equivalent to: `-xmemalign=1i -aligncommon=1`

See `-xmemalign`, page 111.

`-mp={%none|sun|cray|openmp}`

Select the style for parallelization directives.
Fortran parallelization features require a Forte for HPC license.

- **f77/f95**

The default without specifying `-mp` is `%none`.

`-mp=sun`	Accept Sun–style directives: `C$PAR` or `!$PAR` prefix.
`-mp=cray`	Accept Cray–style directives: `CMIC$` or `!MIC$` prefix.
`-mp=openmp`	Accept OpenMP Fortran directives (Available with `f95` only).
`-mp=%none`	Ignore all parallelization directives.

You can combine OpenMP directives with Sun or Cray directives in the same compilation unit. But both Sun and Cray directives cannot both be active in the same compilation unit. For example:

`-mp=sun,openmp` and
`-mp=cray,openmp` are permitted, but `-mp=sun,cray` is not.

You must also specify `-explicitpar` (or `-parallel`) to enable parallelization. For correctness, also specify `-stackvar`:

```
-explicitpar -stackvar -mp=openmp
```

When compiling for OpenMP, use the `-openmp` flag, which includes `-mp=openmp` along with other flags required by OpenMP. See page 79.

A summary of these `f77/f95` directives appears in Appendix E in this manual. See the *Fortran Programming Guide* for details.

`-mt`

Require thread–safe libraries.

- **f77/f95**

Require linking to thread–safe libraries. If you do your own low–level thread management (for example, by calling the `libthread` library), compiling with `–mt` prevents conflicts.

Use `–mt` if you mix Fortran with multithreaded C code that calls the `libthread` library. See also the Solaris *Multithreaded Programming Guide*.

`–mt` is implied automatically when using the `-autopar`, `-explicitpar`, or `-parallel` options.

Note the following:

- A function subprogram that does I/O should not itself be referenced as part of an I/O statement. Such *recursive* I/O may cause the program to deadlock with `–mt`.
- In general, do *not* compile your own multithreaded code with `-autopar`, `-explicitpar`, or `-parallel`. The compiler-generated calls to the threads library and the program's own calls may conflict, causing unexpected results.
- On a single–processor system, performance may be degraded with the `–mt` option.

`-native`

Optimize performance for the host system. (*Obsolete*)

- **f77/f95**

This option is a synonym for `–xtarget=native`. The `–fast` option sets `-xtarget=native`.

-noautopar

Disable automatic parallelization.

- **f77/f95**

  Disables automatic parallelization invoked by `-autopar` earlier on the command line.

-nodepend

Cancel `-depend` in command line.

- **f77/f95**

  Cancel any `-depend` appearing earlier on the command line.

-noexplicitpar

Disable explicit parallelization.

- **f77/f95**

  Disables explicit parallelization invoked by `-explicitpar` earlier on the command line.

-nolib

Disable linking with system libraries.

- **f77/f95**

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

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

  For example, consider a program linked dynamically with `libF77` that fails on a remote system because it has no `libF77`. With this option you can link the library into your program statically.

  Link `libF77` statically and link `libc` dynamically with `f77`:

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

Link `libm` statically and `libc` dynamically with `f95`:

```
demo% f95 -nolib any.f95 -Bstatic -lm -Bdynamic -lc
```

The order for the `-l`*x* options is important. Follow the order shown in the examples.

`-nolibmil`

Cancel `-libmil` on command line.

- **f77/f95**

Use this option *after* the `-fast` option to disable inlining of `libm` math routines:

```
demo% f77 -fast -nolibmil …
```

`-noqueue`

Disable license queueing.

- **f77/f95**

With this option, if no software license is available to run the compiler, it returns without queueing your request and without compiling. A nonzero environment status is returned for testing in `make` files.

`-noreduction`

Disable `-reduction` on command line.

- **f77/f95**

This option disables `-reduction`.

`-norunpath`

Do not build a runtime shared library search path into the executable.

- **f77/f95**

The compiler normally builds into an executable a path that tells the runtime linker where to find the shared libraries it will need. The path is installation dependent. The `-norunpath` option prevents that path from being built in to the executable.

This option is helpful when libraries have been installed in some nonstandard location, and you do not wish to make the loader search down those paths when the executable is run at another site. Compare with `-R`*paths*.

See the *Fortran Programming Guide* chapter on *Libraries* for more information.

`-O`[*n*]

Specify optimization level.

- **f77/f95**

n can be `1`, `2`, `3`, `4`, or `5`. No space is allowed between `-O` and *n*.

If `-O`[*n*] is not specified, only a very basic level of optimization limited to local common subexpression elimination and dead code analysis is performed. A program's performance may be significantly improved when compiled with an optimization level than without optimization. Use of `–O` (which sets `–O3`) or `–fast` (which sets `–O5`) is recommended for most programs.

Each `–O`*n* level includes the optimizations performed at the levels below it. Generally, the higher the level of optimization a program is compiled with, the better runtime performance obtained. However, higher optimization levels may result in increased compilation time and larger executable files.

Debugging with `-g` does not suppress `-O`*n*, but `-O`*n* limits `-g` in certain ways; see the `dbx` documentation.

The `-O3` and `-O4` options reduce the utility of debugging such that you cannot display variables from `dbx`, but you can still use the `dbx` **`where`** command to get a symbolic traceback.

If the optimizer runs out of memory, it attempts to proceed over again at a lower level of optimization, resuming compilation of subsequent routines at the original level.

For details on optimization, see the *Fortran Programming Guide* chapters *Performance Profiling*, and *Performance and Optimization*.

`-O`

This is equivalent to `–O3`.

`-O1`

Provides a minimum of statement–level optimizations.

Use if higher levels result in excessive compilation time, or exceed available swap space.

-O2

Enables basic block level optimizations.

This level usually gives the smallest code size. (See also `-xspace`.)

`-O3` is preferred over `-O2` unless `-O3` results in unreasonably long compilation time, exceeds swap space, or generates excessively large executable files.

-O3

Adds loop unrolling and global optimizations at the function level.

Usually `-O3` generates larger executable files.

-O4

Adds automatic inlining of routines contained in the same file.

Usually `-O4` generates larger executable files due to inlining.

The `-g` option suppresses the `-O4` automatic inlining described above. `-xcrossfile` increases the scope of inlining with `-O4`.

-O5

Attempt aggressive optimizations.

Suitable only for that small fraction of a program that uses the largest fraction of compute time. `-O5`'s optimization algorithms take more compilation time, and may also degrade performance when applied to too large a fraction of the source program.

Optimization at this level is more likely to improve performance if done with profile feedback. See `-xprofile=`*p*.

-o *name*

Specify the name of the executable file to be written.

- **f77/f95**

There must be a blank between `-o` and *name*. Without this option, the default is to write the executable file to `a.out`. When used with `-c`, `-o` specifies the target `.o` object file; with `-G` it specifies the target `.so` library file.

-oldldo

Select an earlier list–directed output style.

- **f77**

Omit the blank that starts each record for list–directed output. This is a change from `f77` releases 1.4 and earlier. The default behavior is to provide that blank, since the Fortran Standard requires it. Note also the `FORM='PRINT'` option of `OPEN`. You can compile parts of a program with `-oldldo` and other parts without it.

-onetrip

Enable one trip `DO` loops.

- **f77/f95**

Compile `DO` loops so that they are executed at least once. `DO` loops in standard Fortran are not performed at all if the upper limit is smaller than the lower limit, unlike some legacy implementations of Fortran.

-openmp

Enable explicit parallelization with Fortran 95 OpenMP directives. *Fortran parallelization features require a Forte for HPC license.*

- **f95**

This option is a macro that combines these options:

```
-mp=openmp -explicitpar -stackvar -D_OPENMP=200011
```

OpenMP directives are summarized in Appendix E.

To run a parallelized program in a multithreaded environment, you must set the `PARALLEL` (or `OMP_NUM_THREADS`) environment variable prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set the `PARALLEL` or `OMP_NUM_THREADS` variable to the available number of processors on the target platform.

OpenMP requires the definition of the preprocessor symbol `_OPENMP` to have the decimal value YYYYMM where YYYY and MM are the year and month designations of the version of the OpenMP Fortran API that the implementation supports.

-p

Compile for profiling with the `prof` profiler.

- **f77/f95**

Prepare object files for profiling, see *prof* (1). If you compile and link in separate steps, and also compile with the `-p` option, then be sure to link with the `-p` option. `-p` with `prof` is provided mostly for compatibility with older systems. `-pg` profiling with `gprof` is possibly a better alternative. See the *Fortran Programming Guide* chapter on *Performance Profiling* for details.

-pad[=*p*]

Insert padding for efficient use of cache.

- **f77/f95**

This option inserts padding between arrays or character variables, if they are static local and not initialized, or if they are in common blocks. The extra padding positions the data to make better use of cache. In either case, the arrays or character variables can not be equivalenced.

p, if present, must be either or both of:

`local`	Add padding between adjacent *local* variables
`common`	Add padding between variables in common blocks

Defaults for `-pad`:

- Without the `-pad`[=*p*] option, the compiler does no padding.
- With `-pad`, but without the =*p*, the compiler does both local and common padding.

The following are equivalent:

- `f77 -pad` *any*`.f`
- `f77 -pad=local,common` *any*`.f`
- `f77 -pad=common,local` *any*`.f`

The `-pad`[=*p*] option applies to items that satisfy the following criteria:

- The items are arrays or character variables
- The items are static local or in common blocks

For a definition of local or static variables, see `-stackvar`, page 88.

Restrictions on `-pad=common`:

- Neither the arrays nor the character strings are equivalenced

- If `–pad=common` is specified for compiling a file that references a common block, it must be specified when compiling all files that reference that common block. The option changes the spacing of variables within the common block. If one program unit is compiled with the option and another is not, references to what should be the same location within the common block might reference different locations.
- If `–pad=common` is specified, the declarations of common block variables in different program units must be the same except for the names of the variables.The amount of padding inserted between variables in a common block depends on the declarations of those variables. If the variables differ in size or rank in different program units, even within the same file, the locations of the variables might not be the same.
- If `–pad=common` is specified, `EQUIVALENCE` declarations involving common block variables are flagged with a warning message and the block is not padded.
- Avoid overindexing arrays in common blocks with `-pad=common` specified. The altered positioning of adjacent data in a padded common block will cause overindexing to fail in unpredictable ways.

`-parallel`

Parallelize with: `–autopar, –explicitpar, –depend`

Fortran parallelization features require a Forte for HPC license.

- **f77/f95**

Parallelize loops chosen automatically by the compiler as well as explicitly specified by user supplied directives. Optimization level is automatically raised to `–O3` if it is lower.

To improve performance, also specify the `–stackvar` option when using any of the parallelization options, including `–autopar`.

Use `-mp`, page 73, to select Sun, Cray, or `f95` OpenMP style parallelization directives.

Avoid `-parallel` if you do your own thread management. See the discussion of `-mt` on page 74.

Parallelization options like `–parallel` are intended to produce executable programs to be run on multiprocessor systems. On a single–processor system, parallelization generally degrades performance.

To run a parallelized program in a multithreaded environment, you must set the `PARALLEL` (or `OMP_NUM_THREADS`) environment variable prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set the `PARALLEL` or `OMP_NUM_THREADS` variable to the available number of processors on the target platform.

If you use `-parallel` and compile and link in *one* step, then linking automatically includes the multithreading library and the thread–safe Fortran runtime library. If you use `-parallel` and compile and link in *separate* steps, then you must also *link* with `-parallel`.

See the *Fortran Programming Guide* chapter *Parallelization* for further information.

-pg

Compile for profiling with the `gprof` profiler.

- **f77/f95**

Compile self–profiling code in the manner of `-p`, but invoke a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file when the program terminates normally. Generate an execution profile by running `gprof`. See the `gprof`(1) man page and the *Fortran Programming Guide* for details.

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

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

-pic

Compile position–independent code for shared library.

- **f77/f95**

Use when compiling dynamic shared libraries. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in program–counter–relative addressing mode through a procedure linkage table.

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

`-pic` is equivalent to `-xcode=pic13`.

There are two nominal performance costs with `-pic` and `-PIC`:

- A routine compiled with either `-pic` or `-PIC` executes a few extra instructions upon entry to set a register to point at the global offset table used for accessing a shared library's global or static variables.
- Each access to a global or static variable involves an extra indirect memory reference through the global offset table. If the compile is done with `-PIC`, there are two additional instructions per global and static memory reference.

When considering the above costs, remember that the use of `-pic` and `-PIC` can significantly reduce system memory requirements, due to the effect of library code sharing. Every page of code in a shared library compiled `-pic` or `-PIC` can be shared by every process that uses the library. If a page of code in a shared library contains even a single non–`pic` (that is, absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether or not a `.o` file has been compiled with `-pic` or `-PIC` is with the `nm` command:

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_
U _GLOBAL_OFFSET_TABLE_
```

A `.o` file containing position–independent code contains an unresolved external reference to `_GLOBAL_OFFSET_TABLE_`, as indicated by the letter `U`.

To determine whether to use `-pic` or `-PIC`, use `nm` to identify the number of distinct global and static variables used or defined in the library. If the size of `_GLOBAL_OFFSET_TABLE_` is under 8,192 bytes, you can use `-pic`. Otherwise, you must use `-PIC`.

When building shared dynamic libraries with `-xarch=v9` (or `v9a` or `v9b`) in 64-bit Solaris environments, the `-pic` or `-PIC` option, or their `-xcode` equivalents, *must* be specified.

-PIC

Compile position–independent code, but with 32-bit addresses.

- **f77/f95**

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

`-PIC` is equivalent to `-xcode=pic32`.

When building shared dynamic libraries with `-xarch=v9` (or `v9a` or `v9b`) in 64-bit Solaris environments, the `-pic` or `-PIC` option, or their `-xcode` equivalents, *must* be specified.

-Qoption *pr ls*

Pass options to compilation phase *pr*.

- **f77/f95**

Pass the suboption list *ls* to the compilation phase *pr*. There must be blanks separating `Qoption`, *pr,* and *ls*. The `Q` can be uppercase or lowercase. The list is a comma–delimited list of suboptions, with no blanks within the list. Each suboption must be appropriate for that program phase, and can begin with a minus sign.

This option is provided primarily for debugging the internals of the compiler by support staff. Use the `LD_OPTIONS` environment variable to pass options to the linker. See the chapter on linking and libraries in the *Fortran Programming Guide.*

-qp

Synonym for `-p`.

- **f77/f95**

-R *ls*

Build dynamic library search paths into the executable file.

- **f77/f95**

With this option, the linker, *ld*(1), stores a list of dynamic library search paths into the executable file.

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

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

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

Use this option to let users run shippable executables without a special path option to find needed dynamic libraries.

Building an executable file using `-R`*paths* adds directory paths to a default path that is always searched last:

Standard Default Path: `/opt/SUNWspro/lib`

For more information, see the *Libraries* chapter in the *Fortran Programming Guide*, and the Solaris *Linker and Libraries Guide*.

-r8

Double default byte size for REAL, INTEGER, DOUBLE and COMPLEX.

- f77

Note – -r8 and -dbl, are now considered obsolete and may be removed in future releases. Use the more general -xtypemap option instead.

-r8 promotes the default byte size for REAL, INTEGER, DOUBLE, and COMPLEX variables declared *without an explicit byte size* as follows:

TABLE 3-9 Default Data Sizes and –r8 (Bytes)

Without –r8 option		With –r8 option
Data Type	**default**	**SPARC**
INTEGER	4	8
REAL	4	8
DOUBLE	8	16

This option applies to variables, parameters, constants, and functions.

Also, LOGICAL is treated as INTEGER, COMPLEX as two REALs, and DOUBLE COMPLEX as two DOUBLEs.

-dbl and –r8 can be expressed in terms of the more general –xtypemap= option:

- -dbl *same as:* -xtypemap=real:64,double:128,integer:64
- -r8 *same as:* -xtypemap=real:64,double:128,integer:mixed

These options promote default DOUBLE PRECISION data to QUAD PRECISION (128 bits). This may be unwanted and may cause performance degradation. It might be more appropriate to use -xtypemap=real:64,double:64,integer:64 instead of -r8 in these cases.

- For all of the floating point data types, -dbl works the same as -r8; using both -r8 and -dbl produces the same results as using only -dbl.
- For INTEGER and LOGICAL data types, -dbl differs from -r8:
  - -dbl allocates 8 bytes, and does 8–byte arithmetic
  - -r8 allocates 8 bytes, and does only 4–byte arithmetic (“mixed”)

In general, if you compile *one* subprogram with -r8, then be sure to compile *all* subprograms of that program with -r8. This also important with programs communicating through unformatted I/O files — if one program is compiled with

`-r8`, then the other program must be similarly compiled with `-r8`. Be also aware that this option alters the default data size of function names, including calls to library functions, unless the function name is typed explicitly with a data size.

The impact on runtime performance may be great. With `–r8`, an expression like `float = 15.0d0*float` is evaluated in quadruple precision due to the declaration of the constant.

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

-r8const

Promote single-precision constants to `REAL*8` constants.

- **f77/f95**

All single-precision `REAL` constants are promoted to `REAL*8`. Double-precision (`REAL*8`) constants are not changed. This option only applies to constants. To promote both constants and variables use `-xtypemap`, page 121.

–reduction

Recognize reduction operations in loops.

- **f77/f95**

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

A *reduction operation* accumulates the elements of an array into a single scalar value. For example, summing the elements of a vector is a typical reduction operation. Although these operations violate the criteria for parallelizability, the compiler can recognize them and parallelize them as special cases when `–reduction` is specified. See the *Fortran Programming Guide* chapter *Parallelization* for information on reduction operations recognized by the compilers.

This option is usable only with the automatic parallelization options `–autopar` or `-parallel`. It is ignored otherwise. Explicitly parallelized loops are not analyzed for reduction operations.

Example: Automatically parallelize with *reduction*:

```
demo% f77 -parallel -reduction any.f
```

-S

Compile and only generate assembly code.

- **f77/f95**

Compile the named programs and leave the assembly–language output on corresponding files suffixed with `.s`. No `.o` file is created.

-s

Strip the symbol table out of the executable file.

- **f77/f95**

This option makes the executable file smaller and more difficult to reverse engineer. However, this option inhibits debugging with `dbx` or other tools, and overrides `-g`.

-sb

Produce table information for the Sun WorkShop source code browser.

- **f77/f95**

See *Using Sun WorkShop* for more information.

Note: `-sb` cannot be used on source files the compiler automatically passes through the `fpp` or `cpp` preprocessors (that is, files with `.F`, `.F90`, or `.F95` extensions), or used with the `-F` option.

-sbfast

Produce *only* source code browser tables.

- **f77/f95**

Produce *only* table information for the Sun WorkShop source code browser and stop. Do not assemble, link, or make object files.

Note: `-sbfast` cannot be used on source files the compiler automatically passes through the `fpp` or `cpp` preprocessors (that is, files with `.F`, `.F90`, or `.F95` extensions), or used with the `-F` option.

-silent

Suppress compiler messages.

- **f77/f95**

Use this option to suppress non–essential messages from the compiler; error and warning messages are still issued. The default is to show file and entry names as they are reached during the compilation.

-stackvar

Force all local variables to be allocated on the memory stack.

- **f77/f95**

Allocate on the memory stack all the *local* variables and arrays in routines, unless otherwise specified. This option makes these variables *automatic*, rather than *static*, and provides more freedom to the optimizer when parallelizing loops with calls to subprograms.

Use of –`stackvar` is recommended with any of the parallelization options.

Variables and arrays are local, unless they are:

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

f77 only: Initializing a local variable in a `DATA` statement after an executable reference to that variable is an extension to `f77`, and is flagged as an error when `-stackvar` is used:

```
demo% cat stak.f
        real x
        x =  1.
        t = 0.
        print*, t
        data x/3.0/
        print *,x+t
        end
demo% f77 -o stak -stackvar stak.f
stak.f:
 MAIN:
"stak.f", line 5: Error: attempt to initialize an automatic
 variable: x
```

Putting large arrays onto the stack with `-stackvar` can overflow the stack causing segmentation faults. Increasing the stack size may be required.

The initial thread executing the program has a *main* stack, while each helper thread of a multithreaded program has its own *thread* stack.

The default stack size is about 8 Megabytes for the main stack and 1 Megabyte (2 Megabytes on SPARC V9 platforms) for each thread stack. The `limit` command (with no parameters) shows the current main stack size. If you get a segmentation fault using `-stackvar`, try increasing the main and thread stack sizes.

Example: Show the current *main* stack size:

```
demo% limit
cputime         unlimited
filesize        unlimited
datasize        523256 kbytes
stacksize       8192 kbytes      <---
coredumpsize    unlimited
descriptors     64
memorysize      unlimited
demo%
```

Example: Set the *main* stack size to 64 Megabytes:

```
demo% limit stacksize 65536
```

Example: Set each *thread* stack size to 8 Megabytes:

```
demo% setenv STACKSIZE 8192
```

For further information of the use of `-stackvar` with parallelization, see the *Parallelization* chapter in the *Fortran Programming Guide.* See `csh`(1) for details on the `limit` command.

`-stop_status=`*yn*

Permit `STOP` statement to return an integer status value.

- **f77/f95**

yn is either `yes` or `no`. The default is `no`.

With `-stop_status=yes`, a `STOP` statement may contain an integer constant. That value will be passed to the environment as the program terminates:

```
STOP 123
```

The value must be in the range 0 to 255. Larger values are truncated and a run–time message issued. Note that

```
STOP 'stop string'
```

is still accepted and returns a status value of 0 to the environment, although a compiler warning message will be issued.

The environment status variable is `$status` for the C shell `csh`, and `$?` for the Bourne and Korn shells, `sh` and `ksh`.

`-temp=`*dir*

Define directory for temporary files.

- **f77/f95**

Set directory for temporary files used by the compiler to be *dir*. No space is allowed within this option string. Without this option, the files are placed in the `/tmp` directory.

`-time`

Time each compilation phase.

- **f77/f95**

The time spent and resources used in each compiler pass is displayed.

`-U`

Recognize upper and lower case in source files.

- **f77/f95**

Do not treat uppercase letters as equivalent to lowercase. The default is to treat uppercase as lowercase except within character–string constants. With this option, the compiler treats `Delta`, `DELTA`, and `delta` as different symbols.

Portability and mixing Fortran with other languages may require use of `-U`. These are discussed in the *Fortran Programming Guide*.

-U*name*

Undefine preprocessor macro *name*.

- **f77/f95**

This option applies only to `.F` and `.F95` source files that invoke the `fpp` or `cpp` preprocessor. It removes any initial definition of the preprocessor macro *name* created by `-D`*name* on the same command line, including those implicitly placed there by the command-line driver, regardless of the order the options appear. It has no effect on any macro definitions in source files. Multiple `-U`*name* flags can appear on the command line. There must be no space between `-U` and the macro *name*.

-u

Report undeclared variables.

- **f77/f95**

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

-unroll=*n*

Enable unrolling of DO loops where possible.

- **f77/f95**

n is a positive integer. The choices are:

- *n*=1 inhibits all loop unrolling.
- *n*>1 *suggests* to the optimizer that it attempt to unroll loops *n* times.

Loop unrolling generally improves performance, but will increase the size of the executable file. For more information on this and other compiler optimizations, see the *Performance and Optimization* chapter in the *Fortran Programming Guide*. See also the discussion of the `UNROLL` directive on page 27.

-V

Show name and version of each compiler pass.

- **f77/f95**

This option prints the name and version of each pass as the compiler executes.

This information may be helpful when discussing problems with Sun service engineers.

-v

Verbose mode – show details of each compiler pass.

- **f77/f95**

  Like `-V`, shows the name of each pass as the compiler executes, and details the options, macro flag expansions, and environment variables used by the driver.

-vax=*v*

Specify choice of VMS Fortran extensions enabled.

- **f77**

v must be a comma–separated list of at least one suboption. Negatives may be constructed by prefixing each suboption keyword by `no%` (as in `no%logical_name`).

The primary options are `-vax=align` and `-vax=misalign`.

`-vax=align` selects all the suboptions without allowing misaligned data. This is the behavior of the `-xl` option prior to `f77` release 3.0.1.

`-vax=misalign` selects all the suboptions and allows misaligned data. This is the behavior of the `-xl` option with `f77` releases 3.0.1, 4.0, 4.2, 5.0, and Sun WorkShop 6.

The table below lists suboptions that can be individually selected.

TABLE 3-10 `-vax=` Suboptions

`-vax=`	Affect
`blank_zero`	Treat blank in a numeric field as zero.
`bslash`	Allow backslash (‘\’) in character constants.
`debug`	Allow VMS Fortran ‘D’ debugging statements.
`logical_name`	Allow VMS Fortran style logical file names.
`oct_const`	Allow double quote character to signify octal constants.
`param`	Allow non–standard form of PARAMETER statement.
`rsize`	Allow unformatted record size in words rather than bytes.
`struct_align`	Align structures as in VMS Fortran.

`%all` and `%none` can also be used to select all or none of these suboptions. Sub–options accumulate from left to right. For example, to enable all but one feature: `-vax=%all,no%rsize` See also `-xl` and `-misalign`.

-vpara

Show verbose parallelization messages.

- **f77/f95**

As the compiler analyzes loops explicitly marked for parallelization with directives, it issues warning messages about certain data dependencies it detects; but the loop will still be parallelized.

Example: `-vpara` for verbose parallelization warnings:

```
demo% f77 -explicitpar -vpara any.f
any.f:
 MAIN any:
"any.f", line 11: Warning: the loop may have parallelization
inhibiting reference
```

-w

Suppress warning messages.

- **f77/f95**

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

Example: `–w` still allows some warnings to get through:

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

For `f95:` Individual levels from 0 to 4 can be specified: `–w0` suppresses the least messages while `–w4` suppresses most warning. `–w` is equivalent to `–w0`.

-Xlist[*x*]

Produce listings and do global program checking (GPC).

- **f77/f95**

Use this option to find potential programming bugs. It invokes an extra compiler pass to check for consistency in subprogram call arguments, common blocks, and parameters, across the global program. The option also generates a line–numbered

listing of the source code, including a cross reference table. The error messages issued by the –`Xlist` options are advisory warnings and do not prevent the program from being compiled and linked.

Note – Be sure to correct all syntax errors in the source code before compiling with `-Xlist`.Unpredictable reports may result when run on a source code with syntax errors.

Example: Check across routines for consistency:

```
demo% f95 -Xlist  fil.f
```

The above example writes the following to the output file `fil.lst`:

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

By default, the listings are written to the file `name.lst`, where `name` is taken from the first listed source file on the command line.

A number of sub–options provide further flexibility in the selection of actions. These are specified by suffixes to the main –`Xlist` option, as shown in the following table

TABLE 3-11 –`Xlist` Suboptions

Option	Feature
–`Xlist`	Show errors, listing, and cross reference table
–`Xlistc`	Show call graphs and errors
–`XlistE`	Show errors
–`Xlisterr`*[nnn]*	Suppress error *nnn* messages
–`Xlistf`	Show errors, listing, and cross references, but no object files
–`Xlistfln`*dir*	Put .fln files in directory *dir*, which must already exist *(*`f77` *only)*
–`Xlisth`	Terminate compilation if errors detected
–`XlistI`	Analyze `#include` and `INCLUDE` files as well as source files
–`XlistL`	Show listing and errors only
–`Xlistl`*n*	Set page length to *n* lines
–`Xlisto` *name*	Rename report file to *name.*`lst`
–`Xlists`	Suppress unreferenced names from the cross–reference table

TABLE 3-11 `-Xlist` Suboptions *(Continued)*

Option	Feature
`-Xlistv`*n*	Set checking level to *n* (1,2,3, or 4) – default is 2
`-Xlistw`*[nnn]*	Set width of output line to *nnn* columns – default is 79
`-Xlistwar`*[nnn]*	Suppress warning *nnn* messages
`-XlistX`	Show cross–reference table and errors

Option `-Xlistfln`*dir* is not available with `f95`.

See the *Fortran Programming Guide* chapter *Program Analysis and Debugging* for details.

`-xa`

Synonym for `-a`.

- **f77/f95**

`-xarch=`*isa*

Specify instruction set architecture (ISA).

Architectures that are accepted by `-xarch` keyword *isa* are shown in TABLE 3-12:

TABLE 3-12 `-xarch` ISA Keywords

Platform	Valid -xarch Keywords
SPARC	`generic, generic64, native, native64, v7, v8a, v8, v8plus, v8plusa, v8plusb, v9, v9a, v9b`

Note that although `-xarch` can be used alone, it is part of the expansion of the `-xtarget` option and may be used to override the `-xarch` value that is set by a specific `-xtarget` option. For example:

```
% f95 -xtarget=ultra2 -xarch=v8plusb ...
```

overrides the `-xarch=v8` set by `-xtarget=ultra2`

This option limits the code generated by the compiler to the instructions of the specified instruction set architecture by allowing only the specified set of instructions. This option does not guarantee use of any target–specific instructions.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice results in a binary program that is not executable on the intended target platform.

TABLE 3-13 summarizes the most general `-xarch` options:

TABLE 3-13 Most General `-xarch` Options on SPARC Platforms

-xarch=	Performance
`generic`	• runs adequately on all platforms
`v8plusa`	• runs optimally on UltraSPARC-II processors in 32-bit mode • no execution on other platforms
`v8plusb`	• runs optimally on UltraSPARC-III processors in 32-bit mode • no execution on other platforms
`v9a`	• runs optimally on UltraSPARC-II processors in 64-bit mode • no execution on other platforms
`v9b`	• runs optimally on UltraSPARC-III processors in 64-bit mode • no execution on other platforms

Also note the following:

- SPARC instruction set architectures V7, V8, and V8a are all binary compatible.
- Object binary files (.o) compiled with `v8plus` and `v8plusa` can be linked and can execute together, but only on a SPARC V8plusa compatible platform.
- Object binary files (.o) compiled with `v8plus`, `v8plusa`, and `v8plusb` can be linked and can execute together, but only on a SPARC V8plusb compatible platform.
- `-xarch` values `v9`, `v9a`, and `v9b` are only available on UltraSPARC 64–bit Solaris environments.
- Object binary files (.o) compiled with `v9` and `v9a` can be linked and can execute together, but will run only on a SPARC V9a compatible platform.
- Object binary files (.o) compiled with `v9`, `v9a`, and `v9b` can be linked and can execute together, but will run only on a SPARC V9b compatible platform.

For any particular choice, the generated executable may run much more slowly on earlier architectures. Also, although quad-precision (`REAL*16` and `long double`) floating-point instructions are available in many of these instruction set architectures, the compiler does not use these instructions in the code it generates.

TABLE 3-14 gives details for each of the `-xarch` keywords on SPARC platforms.

TABLE 3-14 `-xarch` Values for SPARC Platforms

-xarch=	Meaning
`generic`	**Compile for good performance on most 32-bit systems.** This is the default. This option uses the best instruction set for good performance on most processors without major performance degradation on any of them. With each new release, the definition of "best" instruction set may be adjusted, if appropriate, and is currently `v7`.
`generic64`	**Compile for good performance on most 64-bit enabled systems.** This option uses the best instruction set for good performance on most 64-bit enabled processors without major performance degradation on any of them. With each new release, the definition of "best" instruction set may be adjusted, if appropriate, and is currently interpreted as `v9`.
`native`	**Compile for good performance on this system.** This is the default for the `-fast` option. The compiler chooses the appropriate setting for the current system processor it is running on.
`native64`	**Compile for good performance in 64-bit mode on this system.** Like `native`, compiler chooses the appropriate setting for 64-bit mode on the current system processor it is running on.
`v7`	**Compile for the SPARC-V7 ISA.** Enables the compiler to generate code for good performance on the V7 ISA. This is equivalent to using the best instruction set for good performance on the V8 ISA, but without integer `mul` and `div` instructions, and the `fsmuld` instruction. Examples: SPARCstation 1, SPARCstation 2
`v8a`	**Compile for the V8a version of the SPARC-V8 ISA.** By definition, V8a means the V8 ISA, but without the `fsmuld` instruction. This option enables the compiler to generate code for good performance on the V8a ISA. Example: Any system based on the microSPARC I chip architecture
`v8`	**Compile for the SPARC-V8 ISA.** Enables the compiler to generate code for good performance on the V8 architecture. Example: SPARCstation 10

TABLE 3-14 `-xarch` Values for SPARC Platforms *(Continued)*

-xarch=	Meaning
`v8plus`	**Compile for the V8plus version of the SPARC-V9 ISA.** By definition, V8plus means the V9 ISA, but limited to the 32–bit subset defined by the V8plus ISA specification, without the Visual Instruction Set (VIS), and without other implementation-specific ISA extensions. • This option enables the compiler to generate code for good performance on the V8plus ISA. • The resulting object code is in SPARC-V8+ ELF32 format and only executes in a Solaris UltraSPARC environment—it does not run on a V7 or V8 processor. Example: Any system based on the UltraSPARC chip architecture
`v8plusa`	**Compile for the V8plusa version of the SPARC-V9 ISA.** By definition, V8plusa means the V8plus architecture, plus the Visual Instruction Set (VIS) version 1.0, and with UltraSPARC extensions. • This option enables the compiler to generate code for good performance on the UltraSPARC architecture, but limited to the 32–bit subset defined by the V8plus specification. • The resulting object code is in SPARC-V8+ ELF32 format and only executes in a Solaris UltraSPARC environment—it does not run on a V7 or V8 processor. Example: Any system based on the UltraSPARC chip architecture
`v8plusb`	**Compile for the V8plusb version of the SPARC-V8plus ISA with UltraSPARC-III extensions.** Enables the compiler to generate object code for the UltraSPARC architecture, plus the Visual Instruction Set (VIS) version 2.0, and with UltraSPARC-III extensions. • The resulting object code is in SPARC-V8+ ELF32 format and executes only in a Solaris UltraSPARC-III environment. • Compiling with this option uses the best instruction set for good performance on the UltraSPARC-III architecture.
`v9`	**Compile for the SPARC–V9 ISA.** Enables the compiler to generate code for good performance on the V9 SPARC architecture. • The resulting .o object files are in ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9` is only available when compiling in a 64–bit enabled Solaris environment.

TABLE 3-14 `-xarch` Values for SPARC Platforms *(Continued)*

-xarch=	Meaning
`v9a`	**Compile for the SPARC–V9 ISA with UltraSPARC extensions.** Adds to the SPARC-V9 ISA the Visual Instruction Set (VIS) and extensions specific to UltraSPARC processors, and enables the compiler to generate code for good performance on the V9 SPARC architecture. • The resulting .o object files are in ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9a` is only available when compiling in a 64–bit enabled Solaris operating environment.
`v9b`	**Compile for the SPARC-V9 ISA with UltraSPARC-III extensions.** Adds UltraSPARC-III extensions and VIS version 2.0 to the V9a version of the SPARC-V9 ISA. Compiling with this option uses the best instruction set for good performance in a Solaris UltraSPARC-III environment. • The resulting object code is in SPARC-V9 ELF64 format and can only be linked with other SPARC-V9 object files in the same format. • The resulting executable can only be run on an UltraSPARC-III processor running a 64–bit enabled Solaris operating environment with the 64–bit kernel. • `-xarch=v9b` is only available when compiling in a 64–bit enabled Solaris operating environment.

`-xautopar`

Synonym for `-autopar`.

- **f77/f95**

`-xcache=`*c*

Define cache properties for the optimizer.

- **f77/f95**

c must be one of the following:

- `generic`
- *s1*`/`*l1*`/`*a1*
- *s1*`/`*l1*`/`*a1*`:`*s2*`/`*l2*`/`*a2*
- *s1*`/`*l1*`/`*a1*`:`*s2*`/`*l2*`/`*a2*`:`*s3*`/`*l3*`/`*a3*

The *si*`/`*li*`/`*ai* are defined as follows:

si The size of the data cache at level *i*, in kilobytes

li The line size of the data cache at level *i*, in bytes

ai The associativity of the data cache at level *i*

This option specifies the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; it is provided to allow overriding an `-xcache` value implied by a specific `-xtarget` option.

TABLE 3-15 `-xcache` Values

Value	Meaning
`generic`	Define the cache properties for good performance on most SPARC processors without any major performance degradation. This is the default.
s1`/`*l1*`/`*a1*	Define level 1 cache properties.
s1`/`*l1*`/`*a1*`:`*s2*`/`*l2*`/`*a2*	Define levels 1 and 2 cache properties.
s1`/`*l1*`/`*a1*`:`*s2*`/`*l2*`/`*a2*`:`*s3*`/`*l3*`/`*a3*	Define levels 1, 2, and 3 cache properties

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

A Level 1 cache has: 16K bytes, 32 byte line size, 4–way associativity.

A Level 2 cache has: 1024K bytes, 32 byte line size, direct mapping associativity.

-xcg89

Synonym for `-cg89`.

- **f77/f95**

-xcg92

Synonym for `-cg92`.

- **f77/f95**

-xchip=*c*

Specify target processor for the optimizer.

- **f77/f95**

This option specifies timing properties by specifying the target processor.

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; it is provided to allow overriding a `-xchip` value implied by the a specific `-xtarget` option.

Some effects of `-xchip=`*c* are:

- Instruction scheduling
- The way branches are compiled
- Choice between semantically equivalent alternatives

The following table lists the valid `-xchip` values:

TABLE 3-16 Valid `-xchip` Values

Value	Optimize for:
`generic`	most SPARC processors.
`native`	this 32-bit host platform.
`old`	pre–SuperSPARC processors.
`super`	the SuperSPARC processor.
`super2`	the SuperSPARC II processor.
`micro`	the MicroSPARC processor.
`micro2`	the MicroSPARC II processor.
`hyper`	the HyperSPARC processor.
`hyper2`	the HyperSPARC II processor.
`powerup`	the Weitek PowerUp processor.
`ultra`	the UltraSPARC processor.
`ultra2`	the UltraSPARC II processor.
`ultra2e`	the UltraSPARC IIe processor.
`ultra2i`	the UltraSPARC IIi processor.
`ultra3`	the UltraSPARC III processor.

-xcode=*code*

Specify code address space on SPARC platforms.

- **f77/f95**

The values for *code* are:

`abs32`	Generate 32-bit absolute addresses. Code+data+bss size is limited to 2**32 bytes. This is the default on 32-bit platforms: `-xarch=generic, v7, v8, v8a, v8plus, v8plusa`
`abs44`	Generate 44-bit absolute addresses. Code+data+bss size is limited to 2**44 bytes. Available only on 64-bit platforms: `-xarch=v9, v9a`
`abs64`	Generate 64-bit absolute addresses. Available only on 64-bit platforms: `-xarch=v9, v9a`
`pic13`	Generate position-independent code (small model). Equivalent to `-pic`. Permits references to at most 2**11 unique external symbols on 32-bit platforms, 2**10 on 64-bit platforms.
`pic32`	Generate position-independent code (large model). Equivalent to `-PIC`. Permits references to at most 2**30 unique external symbols on 32-bit platforms, 2**29 on 64-bit platforms.

The defaults (not specifying `-xcode=`*code* explicitly) are:

`-xcode=abs32` on SPARC V8 and V7 platforms.
`-xcode=abs64` on SPARC and UltraSPARC V9 (`-xarch=v9` or `v9a`)

When building shared dynamic libraries with `-xarch=v9` or `v9a` and the 64-bit Solaris 7 environment, `-xcode=pic13` or `-xcode=pic32` (or `-pic` or `-PIC`) *must* be specified.

-xcommonchk[={no|yes}]

Enable runtime checking of common block inconsistencies.

- **f77/f95**

This option provides a debug check for common block inconsistencies in programs using `TASK COMMON` and parallelization. (See the discussion of the `TASK COMMON` directive in the *Parallelization* chapter in the *Fortran Programming Guide*.)

The default is `-xcommonchk=no`; runtime checking for common block inconsistencies is disabled because it will degrade performance. Use `-xcommon=yes` only during program development and debugging, and not for production-quality programs.

Compiling with `-xcommonchk=yes` enables runtime checking. If a common block declared in one source program unit as a regular common block appears somewhere else on a `TASK COMMON` directive, the program will stop with an error message indicating the first such inconsistency.

Example: Missing `TASKCOMMON` directive in `tc.f`

```
demo% cat tc.f
                common /x/y(1000)
                do 1 i=1,1000
 1              y(i) = 1.
                call z(57.)
                end
demo% cat tz.f
                subroutine z(c)
                common /x/h(1000)
C$PAR TASKCOMMON X
C$PAR DOALL
                do 1 i=1,1000
1                       h(i) = c* h(i)
                return
                end
demo% f95 -c -O4 -parallel -xcommonchk tc.f
demo% f95 -c -O4 -parallel -xcommonchk tz.f
demo% f95 -o tc -O4 -parallel -xcommonchk tc.o tz.o
demo% tc
ERROR(libmtsk): inconsistent declaration of threadprivate/
taskcommon
   x_: not declared as threadprivate/taskcommon at line 1 of tc.f
demo%
```

`-xcrossfile`[=*n*]

Enable optimization and inlining across source files.

- **f77/f95**

If specified, *n* may be 0, or 1.

Normally, the scope of the compiler's analysis is limited to each separate file on the command line. For example, `-O4`'s automatic inlining is limited to subprograms defined and referenced within the same source file.

With `-xcrossfile`, the compiler analyzes all the files named on the command line as if they had been concatenated into a single source file.

`-xcrossfile` is only effective when used with `-O4` or `-O5`.

Cross–file inlining creates a possible source file interdependence that would not normally be there. If any file in a set of files compiled together with `-xcrossfile` is changed, then all files must be recompiled to insure that the new code is properly inlined. See the discussion of inlining on page 69.

The default, without `-xcrossfile` on the command line, is `-xcrossfile=0`, and no cross-file optimizations are performed. To enable cross-file optimizations, specify `-xcrossfile` (equivalent to `-xcrossfile=1`).

-xdepend

Synonym for `-depend`.

- **f77/f95**

-xexplicitpar

Synonym for `-explicitpar`.

- **f77/f95**

-xF

Allow function–level reordering by the Sun WorkShop Performance Analyzer.

- **f77/f95**

Allow the reordering of functions (subprograms) in the core image using the compiler, the performance analyzer and the linker. If you compile with the `-xF` option, then run the analyzer, you can generate a map file that optimizes the ordering of the functions in memory depending on how they are used together. A subsequent link to build the executable file can be directed to use that map by using the linker `-Mmapfile` option. It places each function from the executable file into a separate section.

Reordering the subprograms in memory is useful only when the application text page fault time is consuming a large percentage of the application time. Otherwise, reordering may not improve the overall performance of the application. The performance analyzer is part of the Sun WorkShop software. See *Using Sun WorkShop* and *Analyzing Program Performance with Sun WorkShop* for further information on the analyzer.

-xhasc[={yes|no}]

Treat Hollerith constant as a character string in an actual argument list.

- **f77/f95**

With `-xhasc=yes`, the compiler treats Hollerith constants as character strings when they appear as an actual argument on a subroutine or function call. This is the default, and complies with the Fortran 77 standard. (The actual call list generated by the compiler contains hidden string lengths for each character string.)

With `-xhasc=no`, Hollerith constants are treated as typeless values in subprogram calls, and only their addresses are put on the actual argument list. (No string length is generated on the actual call list passed to the subprogram.)

Compile routines with `-xhasc=no` if they call a subprogram with a Hollerith constant and the called subprogram expects that argument as `INTEGER` (or anything other than `CHARACTER`).

Example:

```
demo% cat hasc.f
                call z(4habcd, 'abcdefg')
                end
                subroutine z(i, s)
                integer i
                character *(*) s
                print *, "string length = ", len(s)
                return
                end
demo% f77 -o has0 hasc.f
demo% has0
 string length =   4     <-- should be 7
demo% f77 -o has1 -xhasc=no hasc.f
demo% has1
 string length =   7   <-- now correct length for s
```

Passing `4habcd` to `z` is handled correctly by compiling with `-xhasc=no`.

This flag is provided to aid porting older Fortran programs.

-xhelp=*h*

Show summary help information on options or `README` file.

- **f77/f95**

The *h* is either `readme` or `flags`.

`-xhelp=readme` Show the online `README` file for this release of the compiler.
`-xhelp=flags` Show the compiler flags (options), and is same as `-help`.

`-xia[=`*v*`]`

Enable interval arithmetic extensions and set a suitable floating-point environment.

- **f95**

v can be one of either `widestneed` or `strict`. The default if not specified is `widestneed`.

Fortran 95 extensions for interval arithmetic calculations are detailed in the *Interval Arithmetic Programming Reference.* See also `-xinterval`, page 107.

The `-xia` flag is a macro that expands as follows:

`-xia` *or* `-xia=widestneed`	`-xinterval=widestneed -ftrap=%none -fns=no -fsimple=0`
`-xia=strict`	`-xinterval=strict -ftrap=%none -fns=no -fsimple=0`

`-xild{off|on}`

Enable/disable the Incremental Linker.

- **f77/f95**

`-xildoff` disables the use of the incremental linker, `ild`. The standard linker, `ld`, is used instead. `-xildon` enables use of `ild` instead of `ld`.

`-xildoff` is the default if you do *not* use the `-g` option. It is also the default if you use `-G` or name any source file on the command line.

`-xildon` is the default if you use `-g` and do *not* use `-G`, and no source files appear on the command line (just object files and/or libraries).

See the section on `ild` in the *C User's Guide*.

`-xinline=`*list*

Synonym for `-inline`.

- **f77/f95**

-xinterval[=*v*]

Enable interval arithmetic extensions.

- **f95**

v can be one of either `no`, `widestneed` or `strict`. The default if not specified is `widestneed`.

`no`	Interval arithmetic extensions not enabled.
`widestneed`	Promotes all non-interval variables and literals in any mixed-mode expression to the widest interval data type in the expression.
`strict`	Prohibits mixed-type or mixed-length interval expressions. All interval type and length conversions must be explicit.

Fortran 95 extensions for interval arithmetic calculations are detailed in the *Interval Arithmetic Programming Reference.* See also `-xia`, page 106.

-xipo[={0|1}]

Perform interprocedural optimizations.

- **f77/f95**

Performs whole-program optimizations by invoking an interprocedural analysis pass. Unlike `-xcrossfile`, `-xipo` will perform optimizations across all object files in 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 this flag have analysis information compiled within them that enables interprocedural analysis across source and pre-compiled program files. However, analysis and optimization is limited to the object files compiled with `-xipo`, and does not extend to object files on libraries.

`-xipo=0` disables, and `-xipo=1` enables, interprocedural analysis. The default is `-xipo=0`, and if `-xipo` is specified without a value, `-xipo=1` is used.

When compiling and linking are performed in separate steps, `-xipo` must be specified in both steps to be effective.

Example using `-xipo` in a single compile/link step:

```
demo% f95 -xipo -xO4 -o prog  part1.f part2.f part3.f
```

The optimizer performs crossfile inlining across all three source files. This is done in the final link step, so the compilation of the source files need not all take place in a single compilation and could be over a number of separate compilations, each specifying `-xipo`.

Example using `-xipo` in separate compile/link steps:

```
demo% f95 -xipo -xO4 -c part1.f part2.f
demo% f95 -xipo -xO4 -c part3.f
demo% f95 -xipo -xO4 -o prog  part1.o part2.o part3.o
```

The object files created in the compile steps have additional analysis information compiled within them to permit crossfile optimizations to take place at the link step.

A restriction is that libraries, even if compiled with `-xipo` do not participate in crossfile interprocedural analysis, as shown in this example:

```
demo% f95 -xipo -xO4 one.f two.f three.f
demo% ar -r mylib.a one.o two.o three.o
...
demo% f95 -xipo -xO4 -o myprog main.f four.f mylib.a
```

Here interprocedural optimizations will be performed between `one.f`, `two.f` and `three.f`, and between `main.f` and `four.f`, but not between `main.f` or `four.f` and the routines on `mylib.a`. (The first compilation may generate warnings about undefined symbols, but the interprocedural optimizations will be performed because it is a compile and link step.)

Other important information about `-xipo`:

- requires at least optimization level `-xO4`
- conflicts with `-xcrossfile`; if used together will result in a compilation error
- objects compiled without `-xipo` can be linked freely with objects compiled with `-xipo`.
- The `-xipo` option generates significantly larger object files due to the additional information needed to perform optimizations across files. However, this additional information does not become part of the final executable binary file. Any increase in the size of the executable program will be due to the additional optimizations performed
- In this release, crossfile subprogram inlining is the only interprocedural optimization performed by `-xipo`.

-xl[d]

Enable more VMS Fortran extensions.

- **f77**

  `-xl:` Enable the compiler to accept more VMS Fortran extensions. This is a macro that is translated to `-vax=misalign`, and provides the language features that are listed later in this description. See the description of `-vax=`, page 92.

  Although most VMS features are accepted automatically by `f77` without any special options, you must use the `-xl` option for a few VMS extensions.

  In general, you need the `-xl` option if a source statement can be interpreted as either a VMS feature or an `f77` or `f95` feature, and you want the VMS feature. In this case, the `-xl` option forces the compiler to interpret it the VMS way.

  This option enables the following VMS language features:

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

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

  Use `-xld` to cause compilation of debugging comments (`D` or `d` in column one). Without the `-xld` option, they remain comments only. (There is no space between `-xl` and `d`.)

  Programs that share structures with C routines should not use `-xl`.

  See the *Fortran Library Reference* for information on the VMS libraries. See also the chapter on VMS language extensions in the *Fortran 77 Language Reference* that the `f77` compiler automatically recognizes.

-xlang=*pl*

- **f95**

  Prepare for linking with runtime libraries for programming language *pl*.

  For `f95`, only `-xlang=f77` is allowed.

  `f95 -xlang=f77` implies linking with the `f77compat` library, and is a shorthand way for linking Fortran 95 object files with Fortran 77 object files that insures the proper runtime environment.

Use `f95 -xlang=f77` when linking `f95` and `f77` compiled objects together into a single executable.

-xlibmil

Synonym for `-libmil`.

- **f77/f95**

-xlibmopt

Use library of optimized math routines.

- **f77/f95**

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

-xlic_lib=sunperf

Link with the Sun Performance Library.

- **f77/f95**

For example:

```
f77 -o pgx -fast pgx.f -xlic_lib=sunperf
```

As with `-l`, this option should appear on the command line after all source and object file names.

This option must be used to link with the Sun Performance Library. (See the *Sun Performance Library User's Guide*.)

-xlicinfo

Show license server information.

- **f77/f95**

Use this option to return license information about the licensing system—in particular, the name of the license server and the user ID for each of the users who have licenses checked out.

Generally, with this option, no compilation takes place, and a license is not checked out. This option is normally used alone with no other options. However, if a conflicting option is used, then the last one on the command line prevails, and there is a warning.

-xloopinfo

Synonym for `-loopinfo`.

- **f77/f95**

-xmaxopt[=*n*]

Enable optimization pragma and set maximum optimization level.

- **f77/f95**

n has the value 1 through 5 and corresponds to the optimization levels of `-O1` through `-O5`. If not specified, the compiler uses 5.

This option enables the `C$PRAGMA SUN OPT=`*n* directive (see page 28) when it appears in the source input. Without this option, the compiler treats these lines as comments.

If such a pragma directive appears with an optimization level greater than the maximum level on the `-xmaxopt` flag, the compiler uses the level set by `-xmaxopt`.

-xmemalign[=<*a*><*b*>]

Specify maximum assumed memory alignment and behavior of misaligned data accesses.

- **f77/f95**

For memory accesses where the alignment is determinable at compile time, the compiler will generate the appropriate load/store instruction sequence for that data alignment.

For memory accesses where the alignment cannot be determined at compile time, the compiler must assume an alignment to generate the needed load/store sequence.

The `-xmemalign` flag allows the user to specify the maximum memory alignment of data to be assumed by the compiler for those indeterminate situations. It also specifies the error behavior at runtime when a misaligned memory access does take place.

The value specified consists of two parts: a numeric alignment value, <*a*>, and an alphabetic behavior flag, <*b*>.

Allowed values for alignment, <*a*>, are:

`1`	Assume at most 1-byte alignment.
`2`	Assume at most 2-byte alignment.
`4`	Assume at most 4-byte alignment.
`8`	Assume at most 8-byte alignment.
`16`	Assume at most 16-byte alignment.

Allowed values for error behavior on accessing misaligned data, <*b*>, are:

`i`	Interpret access and continue execution
`s`	Raise signal SIGBUS
`f`	Raise signal SIGBUS only for alignments less or equal to 4

The defaults without `-xmemalign` specified are:

- `4s` for `-xarch=generic,v7,v8,v8a,v8plus,v8plusa`
- `8s` for `-xarch=v9,v9a` for C and C++
- `8f` for `-xarch=v9,v9a` for Fortran

The default for `-xmemalign` appearing without a value is `1i` for all platforms.

The `-dalign` (page 52) and `-misalign` (page 73) options are macros:

`-dalign` is a macro for: `-xmemalign=8s -aligncommon=16`
`-misalign` is a macro for: `-xmemalign=1i -aligncommon=1`

-xnolib

Synonym for `-nolib`.

- **f77/f95**

-xnolibmil

Synonym for `-nolibmil`.

- **f77/f95**

-xnolibmopt

Do not use fast math library.

- **f77/f95**

Use with `-fast` to override linking the optimized math library:

```
f77 -fast -xnolibmopt …
```

-xO*n*

Synonym for -O*n*.

- **f77/f95**

-xopenmp

Synonym for -openmp.

- **f95**

-xpad

Synonym for -pad.

- **f77**

-xparallel

Synonym for -parallel.

- **f77/f95**

-xpg

Synonym for -pg.

- **f77/f95**

-xpp={fpp|cpp}

Select source file preprocessor.1fs

- **f77/f95**

The default is -xpp=fpp.

The compilers use fpp(1) to preprocess .F or .f95 source files. This preprocessor is appropriate for Fortran. Previous versions used the standard C preprocessor cpp. To select cpp, specify -xpp=cpp.

-xprefetch[=*a*[,*a*]]

Enable prefetch instructions on platforms that support prefetch, such as UltraSPARC II.

- **f77/f95**

See page 29 for a description of the Fortran PREFETCH directives.

Enable prefetch instructions on those architectures that support prefetch, such as UltraSPARC II (-xarch=v8plus, v8plusa, v9plusb, v9, v9a, or v9b)

a must be one of the following values.

Value	Meaning
auto	Enable automatic generation of prefetch instructions
no%auto	Disable automatic generation of prefetch instructions
explicit	Enable explicit prefetch macros
no%explicit	Disable explicit prefetch macros
latx:*factor*	Adjust the compiler's assumed prefetch-to-load and prefetch-to-store latencies by the specified factor. The factor must be a positive floating-point or integer number.
yes	-xprefetch=yes is the same as -xprefetch=auto,explicit
no	-xprefetch=no is the same as -xprefetch=no%auto,no%explicit

With -xprefetch, -xprefetch=auto, and -xprefetch=yes, the compiler is free to insert prefetch instructions into the code it generates. This may result in a performance improvement on architectures that support prefetch.

If you are running computationally intensive codes on large multiprocessors, you might find it advantageous to use -xprefetch=latx:*factor*. This option instructs the code generator to adjust the default latency time between a prefetch and its associated load or store by the specified factor.

The prefetch latency is the hardware delay between the execution of a prefetch instruction and the time the data being prefetched is available in the cache. The compiler assumes a prefetch latency value when determining how far apart to place a prefetch instruction and the load or store instruction that uses the prefetched data.

Note – The assumed latency between a prefetch and a load may not be the same as the assumed latency between a prefetch and a store.

The compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications. This tuning may not always be optimal. For memory-intensive applications, especially applications intended to run on large multiprocessors, you may be able to obtain better performance by increasing the prefetch latency values. To increase the values, use a factor that is greater than 1. A value between .5 and 2.0 will most likely provide the maximum performance.

For applications with datasets that reside entirely within the external cache, you may be able to obtain better performance by decreasing the prefetch latency values. To decrease the values, use a factor that is less than 1.

To use the `-xprefetch=latx:`*factor* option, start with a factor value near 1.0 and run performance tests against the application. Then increase or decrease the factor, as appropriate, and run the performance tests again. Continue adjusting the factor and running the performance tests until you achieve optimum performance. When you increase or decrease the factor in small steps, you will see no performance difference for a few steps, then a sudden difference, then it will level off again.

Defaults:

If `-xprefetch` is not specified, `-xprefetch=no%auto,explicit` is assumed.

If only `-xprefetch` is specified, `-xprefetch=auto,explicit` is assumed.

The default of `no%auto` is assumed unless explicitly overridden with the use of `-xprefetch` without any arguments or with an argument of `auto` or `yes`. For example, `-xprefetch=explicit` is the same as `-xprefetch=explicit,no%auto`.

The default of `explicit` is assumed unless explicitly overridden with an argument of `no%explicit` or an argument of `no`. For example, `-xprefetch=auto` is the same as `-xprefetch=auto,explicit`.

If automatic prefetching is enabled, such as with `-xprefetch` or `-xprefetch=yes`, but a latency factor is not specified, then `-xprefetch=latx:1.0` is assumed.

Interactions:

With `-xprefetch=explicit`, the compiler will recognize the directives:

```
$PRAGMA SPARC_PREFETCH_READ_ONCE (name)
$PRAGMA SPARC_PREFETCH_READ_MANY (name)
$PRAGMA SPARC_PREFETCH_WRITE_ONCE (name)
$PRAGMA SPARC_PREFETCH_WRITE_MANY (name)
```

The `-xchip` setting effects the determination of the assumed latencies and therefore the result of a `latx:`*factor* setting.

The `latx:`*factor* suboption is valid only when automatic prefetching is enabled. That is, `latx:`*factor* is ignored unless it is used with `yes` or `auto`.

Warnings:

Explicit prefetching should only be used under special circumstances that are supported by measurements.

Because the compiler tunes the prefetch mechanism for optimal performance across a wide range of machines and applications, you should only use `-xprefetch=latx:`*factor* when the performance tests indicate there is a clear benefit. The assumed prefetch latencies may change from release to release. Therefore, retesting the effect of the latency factor on performance whenever switching to a different release is highly recommended.

`-xprofile=`*p*

Collect or optimize with runtime profiling data.

- **f77/f95**

p must be one of `collect[:`*name*`]`, `use[:`*name*`]`, or `tcov`. Optimization level must be `-O2` or greater.

`collect[:`*name*`]`

Collect and save execution frequency data for later use by the optimizer with `-xprofile=use`. The compiler generates code to measure statement execution frequency.

The *name* is the name of the program that is being analyzed. This name is optional. If *name* is not specified, `a.out` is assumed to be the name of the executable.

At runtime a program compiled with `–xprofile=collect:`*name* will create by default the subdirectory *name*`.profile` to hold the runtime feedback information. The program writes its runtime profile data to the file `feedback` in this subdirectory. If you run the program several times, the execution frequency data accumulates in the `feedback` file; that is, output from prior runs is not lost.

You can set the environment variables `SUN_PROFDATA` and `SUN_PROFDATA_DIR` to control the file and directory where a program compiled with `-xprofile=collect` writes its runtime profile data. With these variables set, the program compiled with `-xprofile=collect` writes its profile data to `$SUN_PROFDATA_DIR/$SUN_PROFDATA`.

These environment variables similarly control the path and names of the profile data files written by `tcov`, as described in the `tcov`(1) man page.

`use[:`*nm*`]`

Use execution frequency data to optimize strategically.

As with `collect:nm`, the *nm* is optional and may be used to specify the name of the program.

The program is optimized by using the execution frequency data previously generated and saved in the `feedback` files written by a previous execution of the program compiled with `–xprofile=collect`.

The source files and other compiler options must be exactly the same as used for the compilation that created the compiled program that generated the `feedback` file. If compiled with `–xprofile=collect:`*nm*, the same program name *nm* must appear in the optimizing compilation: `–xprofile=use:`*nm*.

`tcov`

Basic block coverage analysis using "new" style `tcov`.

Code instrumentation is similar to that of `–a`, but `.d` files are no longer generated for each source file. Instead, a single file is generated, whose name is based on the name of the final executable. For example, if `stuff` is the executable file, then `stuff.profile/tcovd` is the data file.

When running `tcov`, you must pass it the `–x` option to make it use the new style of data. If not, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output.

Unlike `–a`, the `TCOVDIR` environment variable has no effect at compile–time. However, its value is used at program runtime to identify where to create the profile subdirectory.

See the `tcov`(1) man page, the *Performance Profiling* chapter of the *Fortran Programming Guide*, and the *Analyzing Program Performance with Sun WorkShop* manual for more details.

Note: The report produced by `tcov` can be unreliable if there is inlining of subprograms due to `-O4` or `-inline`. Coverage of calls to routines that have been inlined is not recorded.

-xrecursive

Allow routines without RECURSIVE attribute call themselves recursively.

- **f95**

  Only subprograms defined with the RECURSIVE attribute can call themselves recursively, unless they are compiled with -xrecursive.

  However, compiling with -xrecursive may cause performance degradations. Also consider using -stackvar with -xrecursive since -xrecursive does not by itself allocate local variables on the memory stack.

-xreduction

Synonym for -reduction.

- **f77/f95**

-xregs=*r*

Specify register usage.

- **f77/f95**

  r is a comma–separated list that consists of one or more of the following:

  [no%]appl, [no%]float.

  Where the % is shown, it is a required character.

  Example: -xregs=appl,no%float

  - appl: Allow using the application registers.

    On SPARC systems, certain registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.

    The set of application registers depends on the SPARC platform:

    - -xarch=v8 or v8a — registers %g2, %g3, and %g4
    - -xarch=v8 or v8a — registers %g2, %g3, and %g4
    - -xarch=v8plus or v8plusa — registers %g2, %g3, and %g4
    - -xarch=v9 or v9a — registers %g2 and %g3

  - no%appl: Do not use the appl registers.
  - float: Allow using the floating–point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating–point code.

- `no%float`: Do not use the floating–point registers. With this option, a source program cannot contain any floating–point code.

The default is: `-xregs=appl,float`.

-xs

Allow debugging by `dbx` without object (`.o`) files .

- **f77/f95**

With `-xs`, if you move executables to another directory, then you can use `dbx` and ignore the object (`.o`) files. Use this option when you cannot keep the `.o` files.

- The compiler passes `-s` to the assembler and then the linker places all symbol tables for `dbx` in the executable file.
- This way of handling symbol tables is the older way. It is sometimes called *no auto–read*
- The linker links more slowly, and `dbx` initializes more slowly.

Without `-xs`, if you move the executables, you must move both the source files and the object (`.o`) files, or set the path with either the dbx `pathmap` or `use` command.

- This way of handling symbol tables is the newer and default way of loading symbol tables. It is sometimes called *auto–read*.
- The symbol tables are distributed in the `.o` files so that `dbx` loads the symbol table information only if and when it is needed. Hence, the linker links faster, and `dbx` initializes faster.

-xsafe=mem

Allow the compiler to assume that no memory protection violations occur.

- **f77/f95**

Using this option allows the compiler to assume no memory–based traps occur. It grants permission to use the speculative load instruction on the SPARC V9 platforms.

This option is effective only when used with optimization level `-O5` one one of the following architectures (`-xarch`): v8plus, v8plusa, v8plusb, v9, v9a, or v9b

Warning:

- Because non-faulting loads do not cause a trap when a fault such as address misalignment or segmentation violation occurs, you should use this option only for programs in which such faults cannot occur. Because few programs incur memory-based traps, you can safely use this option for most programs. Do not use this option with programs that explicitly depend on memory-based traps to handle exceptional conditions.

-xsb

Synonym for `-sb`.

- **f77/f95**

-xsbfast

Synonym for `-sbfast`.

- **f77/f95**

-xspace

Do not allow optimizations to increase code size.

- **f77/f95**

Do no optimizations that increase the code size.

Example: Do not unroll or parallelize loops if it increases code size.

-xtarget=*t*

Specify target platform for optimization.

- **f77/f95**

Specify the target platform for the instruction set and optimization.

t must be one of: `native`, `native64`, `generic`, `generic64`, *platform–name*.

The `-xtarget` option permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on real platforms. The only meaning of `-xtarget` is in its expansion.

The performance of some programs may benefit by providing the compiler with 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 is negligible and a `generic` specification is sufficient.

`native`: Optimize performance for the host platform.

The compiler generates code optimized for the host platform. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.

`native64`: Compile for native 64-bit environment.

Set the architecture, chip, and cache properties for the 64-bit environment on the machine on which the compiler is running.

`generic`: Get the best performance for generic architecture, chip, and cache.

The compiler expands `–xtarget=generic` to:

```
–xarch=generic –xchip=generic –xcache=generic
```

This is the default value.

`generic64`: Compile for generic 64-bit environment.

This expands to `-xarch=v9 -xcache=generic -xchip=generic`

platform–name: Get the best performance for the specified platform.

Appendix D gives a complete list of current SPARC platform names accepted by the compilers. For example, `-xtarget=ultra2i`

`–xtime`

Synonym for `–time`.

- **f77/f95**

`–xtypemap=`*spec*

Specify default data mappings.

- **f77/f95**

This option provides a flexible way to specify the byte sizes for default data types. Use of this option is preferred over `–dbl` and `–r8`, and applies to both default-size variables and constants.

The specification string *spec* may contain any or all of the following in a comma-delimited list:

`real:`*size*
`double:`*size*
`integer:`*size*

The accepted data *size* values are: `64`, `128`, for `real` and `double`; `32`, `64`, and `mixed` for `integer`. For example:

```
-xtypemap=real:64,double:128,integer:64
```

This option applies to all variables declared with default specifications (without explicit byte sizes), as in `REAL XYZ` (resulting in a 64-bit `XYZ`). Also, all single-precision `REAL` constants are promoted to `REAL*8`.

The allowable combinations on each platform are:

- `real:32`
- `real:64`
- `double:64`
- `double:128`
- `integer:32`
- `integer:64`
- `integer:mixed` *(f77 only)*

The `integer:mixed` mapping specifies 8-byte data but only 4-byte arithmetic, and is only available with `f77`. Preferred is `integer:64`.

The `f77` flags `-dbl` and `-r8` options have their `-xtypemap` equivalents:

- `-dbl` *same as:* `-xtypemap=real:64,double:128,integer:64`
- `-r8` *same as:* `-xtypemap=real:64,double:128,integer:mixed`

There are two additional possibilities on

- `-xtypemap=real:64,double:64,integer:mixed`
- `-xtypemap=real:64,double:64,integer:64`

which map both default `REAL` and `DOUBLE` to 8 bytes, and may be preferable over the use of `-dbl` or `-r8` because they do not promote `DOUBLE PRECISION` to `QUAD PRECISION`.

Note that `INTEGER` and `LOGICAL` are treated the same, and `COMPLEX` is mapped as two `REAL`s. Also, `DOUBLE COMPLEX` will be treated the way `DOUBLE` is mapped.

`-xunroll=`*n*

Synonym for `-unroll=`*n*.

- **f77/f95**

`-xvector[={yes|no}]`

Enable automatic calls to the SPARC vector library functions.

- **f77/f95**

With `-xvector=yes`, the compiler is permitted to transform certain math library calls within DO loops into single calls to the equivalent vectorized library routine whenever possible. This could result in a performance improvement for loops with large loop counts.

The compiler defaults to `-xvector=no`. Specifying `-xvector` by itself defaults to `-xvector=yes`.

This option also triggers `-depend`. (Follow `-xvector` with `-nodepend` on the command line to cancel the dependency analysis.)

The compiler will automatically notify the linker to include the `libmvec` and `libc` libraries in the load step if `-xvector` appears. However, to compile and link in separate steps requires specifying `-xvector` on the link step as well to correctly select these necessary libraries.

-xvpara

Synonym for `-vpara`.

- **f77**

-Zlp

Obsolete: Compile for loop performance profiling by `looptool`.

- **f77/f95**

This option and `looptool` are no longer supported. Use the Sun WorkShop Performance Analyzer instead.

Refer to *Analyzing Program Performance With Sun WorkShop,* and the `analyzer`(1) man page for more information.

-ztext

Generate only pure libraries with no relocations.

- **f77/f95**

Do not make the library if relocations remain.

The general purpose of `-ztext` is to verify that a generated library is pure text; instructions are all position–independent code. Therefore, it is generally used with both `-G` and `-pic`.

With `-ztext`, if `ld` finds an incomplete relocation in the *text* segment, then it does not build the library. If it finds one in the *data* segment, then it generally builds the library anyway; the data segment is writable.

Without `-ztext`, `ld` builds the library, relocations or not.

A typical use is to make a library from both source files and object files, where you do not know if the object files were made with `-pic`.

Example: Make library from both source and object files:

```
demo% f77 –G –pic –ztext –o MyLib –hMyLib a.f b.f x.o y.o
```

An alternate use is to ask if the code is position–independent already: compile without `–pic`, but ask if it is pure text.

Example: Ask if it is pure text already—even without `–pic`:

```
demo% f77 –G –ztext –o MyLib –hMyLib a.f b.f x.o y.o
```

If you compile with `–ztext` and `ld` does not build the library, then you can recompile without `–ztext`, and `ld` will build the library. The failure to build with `-ztext` means that one or more components of the library cannot be shared; however, maybe some of the other components can be shared. This raises questions of performance that are best left to you, the programmer.

APPENDIX **A**

Runtime Error Messages

This appendix describes the error messages generated by the Fortran I/O library, signal handler, and operating system.

Operating System Error Messages

Operating system error messages include system call failures, C library errors, and shell diagnostics. The system call error messages are found in `intro`(2). System calls made through the `Fortran` library do not produce error messages directly. The following system routine in the `Fortran` library calls C library routines which produce an error message:

```
    integer system, status
    status = system("cp  afile bfile")
    print*, "status = ", status
    end
```

The following message is displayed:

```
cp: cannot access afile
 status =  512
```

Signal Handler Error Messages (f77)

Before beginning execution of a program, the Fortran 77 library sets up a signal handler (`sigdie`) for signals that can cause termination of the program. `sigdie` prints a message that describes the signal, flushes any pending output, and generates a core image.

Presently, the only arithmetic exception that produces an error message is the `INTEGER*2` division with a denominator of zero. All other arithmetic exceptions are ignored.

A signal handler error example follows, where the subroutine `SUB` tries to access parameters that are not passed to it:

```
    CALL SUB()
    END
    SUBROUTINE SUB(I,J,K)
    I=J+K
    RETURN
    END
```

The following error message results when compiled with `f77` and run:

```
*** TERMINATING  sub77
*** Received signal 11 SIGSEGV
Segmentation Fault
```

The Fortran 95 compiler does not set up error handlers.

I/O Error Messages (f77)

The error messages in this section are generated by the Fortran 77 I/O library. The error numbers are returned in the `IOSTAT` variable if the `ERR` return is taken.

For example, the following program tries to do an unformatted write to a file opened for formatted output:

```
      WRITE( 6 ) 1
      END
```

and produces error messages like the following:

```
sue: [1003] unformatted io not allowed
logical unit 6, named 'stdout'
lately: writing sequential unformatted external IO
```

The following error messages are generated. These same messages are also documented at the end of the man page `perror`(3F).

If the error number is less than 1000, then it is a *system* error. See `intro`(2).

TABLE A-1 `f77` Runtime I/O Messages

Error	Message
1000	`error in format` Read the error message output for the location of the error in the format. It can be caused by more than 10 levels of nested parentheses or an extremely long format statement.
1001	`illegal unit number` It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is 2^{31} - 1.
1002	`formatted io not allowed` The logical unit was opened for unformatted I/O.
1003	`unformatted io not allowed` The logical unit was opened for formatted I/O.
1004	`direct io not allowed` The logical unit was opened for sequential access, or the logical record length was specified as 0.
1005	`sequential io not allowed` The logical unit was opened for direct access I/O.
1006	`can't backspace file` You cannot do a `seek` on the file associated with the logical unit; therefore, you cannot backspace. The file may be a `tty` device or a pipe.
1007	`off beginning of record` You tried to do a left tab to a position before the beginning of an internal input record.

TABLE A-1 `f77` Runtime I/O Messages *(Continued)*

Error	Message
1008	`can't stat file` The system cannot return status information about the file. Perhaps the directory is unreadable.
1009	`no * after repeat count` Repeat counts in list-directed I/O must be followed by an * with no blank spaces.
1010	`off end of record` A formatted write tried to go beyond the logical end-of-record. An unformatted read or write also causes this
1011	*<Not used>*
1012	`incomprehensible list input` List input has to be as specified in the declaration.
1013	`out of free space` The library dynamically creates buffers for internal use. You ran out of memory for them; that is, your program is too big.
1014	`unit not connected` The logical unit was not open.
1015	`read unexpected character` Certain format conversions cannot tolerate nonnumeric data.
1016	`illegal logical input field` logical data must be T or F.
1017	`'new' file exists` You tried to open an existing file with status='new'.
1018	`can't find 'old' file` You tried to open a nonexistent file with status='old'.
1019	`unknown system error` This error should not happen, but...
1020	`requires seek ability` Attempted a `seek` on a file that does not allow it. I/O operation requiring a `seek` are direct access, sequential unformatted I/O, and tabbing left.
1021	`illegal argument` Certain arguments to open and related functions are checked for legitimacy. Often only nondefault forms are checked
1022	`negative repeat count` The repeat count for list-directed input must be a positive integer.
1023	`illegal operation for unit` Attempted an I/O operation that is not possible for the device associated with the logical unit. You get this error if you try to read past end-of-tape, or end-of-file.

TABLE A-1 `f77` Runtime I/O Messages *(Continued)*

Error	Message
1024	*<Not used>*
1025	`incompatible specifiers in open` Attempted to open a file with the 'new' option and the access='append' option, or some other invalid combination.
1026	`illegal input for namelist` A namelist read encountered an invalid data item.
1027	`error in FILEOPT parameter` The `FILEOPT` string in an `OPEN` statement has bad syntax.
1028	`WRITE to readonly file` Attempt to write on a unit that was opened for reading only.
1029	`READ from writeonly file` Attempt to read from a unit that was opened for writing only.
1030	`overflow converting numeric input` Integer input data is too large for the corresponding input variable
1032	`exponent overflow on numeric input` The floating-point input data is too large to be represented by the corresponding input variable.

I/O Error Messages (f95)

Error messages generated by Fortran 95 programs are different than those generated by Fortran 77 programs. Here is the previous example, compiled and run with Fortran 95:

```
demo% cat wf.f
      WRITE( 6 ) 1
      END
demo% f95 -o wf wf.f
demo% wf

 ******  FORTRAN RUN-TIME SYSTEM  ******
 Error 1003:  unformatted I/O on formatted unit
 Location:  the WRITE statement at line 1 of "wf.f"
 Unit:  6
 File:  standard output
Abort
```

Because the `f95` message contains references to the originating source code filename and line number, application developers should consider using the `ERR=` clause in I/O statements to softly trap runtime I/O errors.

TABLE A-2 lists the runtime I/O messages issued by `f95`.

TABLE A-2 `f95` Runtime I/O Messages

Error	Message
1000	format error
1001	illegal unit number
1002	formatted I/O on unformatted unit
1003	unformatted I/O on formatted unit
1004	direct-access I/O on sequential-access unit
1005	sequential-access I/O on direct-access unit
1006	device does not support BACKSPACE
1007	off beginning of record
1008	can't stat file
1009	no * after repeat count

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1010	record too long
1011	truncation failed
1012	incomprehensible list input
1013	out of free space
1014	unit not connected
1015	read unexpected character
1016	illegal logical input field
1017	'new' file exists
1018	can't find 'old' file
1019	unknown system error
1020	requires seek ability
1021	illegal argument
1022	negative repeat count
1023	illegal operation for channel or device
1024	reentrant I/O
1025	incompatible specifiers in open
1026	illegal input for namelist
1027	error in FILEOPT parameter
1028	writing not allowed
1029	reading not allowed
1030	integer overflow on input
1031	floating-point overflow on input
1032	floating-point underflow on input
1051	default input unit closed
1052	default output unit closed
1053	direct-access READ from unconnected unit
1054	direct-access WRITE to unconnected unit
1055	unassociated internal unit
1056	null reference to internal unit

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1057	empty internal file
1058	list-directed I/O on unformatted unit
1059	namelist I/O on unformatted unit
1060	tried to write past end of internal file
1061	unassociated ADVANCE specifier
1062	ADVANCE specifier is not 'YES' or 'NO'
1063	EOR specifier present for advancing input
1064	SIZE specifier present for advancing input
1065	negative or zero record number
1066	record not in file
1067	corrupted format
1068	unassociated input variable
1069	more I/O-list items than data edit descriptors
1070	zero stride in subscript triplet
1071	zero step in implied DO-loop
1072	negative field width
1073	zero-width field
1074	character string edit descriptor reached on input
1075	Hollerith edit descriptor reached on input
1076	no digits found in digit string
1077	no digits found in exponent
1078	scale factor out of range
1079	digit equals or exceeds radix
1080	unexpected character in integer field
1081	unexpected character in real field
1082	unexpected character in logical field
1083	unexpected character in integer value
1084	unexpected character in real value
1085	unexpected character in complex value

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1086	unexpected character in logical value
1087	unexpected character in character value
1088	unexpected character before NAMELIST group name
1089	NAMELIST group name does not match the name in the program
1090	unexpected character in NAMELIST item
1091	unmatched parenthesis in NAMELIST item name
1092	variable not in NAMELIST group
1093	too many subscripts in NAMELIST object name
1094	not enough subscripts in NAMELIST object name
1095	zero stride in NAMELIST object name
1096	empty section subscript in NAMELIST object name
1097	subscript out of bounds in NAMELIST object name
1098	empty substring in NAMELIST object name
1099	substring out of range in NAMELIST object name
1100	unexpected component name in NAMELIST object name
1111	unassociated ACCESS specifier
1112	unassociated ACTION specifier
1113	unassociated BINARY specifier
1114	unassociated BLANK specifier
1115	unassociated DELIM specifier
1116	unassociated DIRECT specifier
1117	unassociated FILE specifier
1118	unassociated FMT specifier
1119	unassociated FORM specifier
1120	unassociated FORMATTED specifier
1121	unassociated NAME specifier
1122	unassociated PAD specifier
1123	unassociated POSITION specifier
1124	unassociated READ specifier

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1125	unassociated READWRITE specifier
1126	unassociated SEQUENTIAL specifier
1127	unassociated STATUS specifier
1128	unassociated UNFORMATTED specifier
1129	unassociated WRITE specifier
1130	zero length file name
1131	ACCESS specifier is not 'SEQUENTIAL' or 'DIRECT'
1132	ACTION specifier is not 'READ', 'WRITE' or 'READWRITE'
1133	BLANK specifier is not 'ZERO' or 'NULL'
1134	DELIM specifier is not 'APOSTROPHE', 'QUOTE', or 'NONE'
1135	unexpected FORM specifier
1136	PAD specifier is not 'YES' or 'NO'
1137	POSITION specifier is not 'APPEND', 'ASIS', or 'REWIND'
1138	RECL specifier is zero or negative
1139	no record length specified for direct-access file
1140	unexpected STATUS specifier
1141	status is specified and not 'OLD' for connected unit
1142	STATUS specifier is not 'KEEP' or 'DELETE'
1143	status 'KEEP' specified for a scratch file
1144	impossible status value
1145	a file name has been specified for a scratch file
1146	attempting to open a unit that is being read from or written to
1147	attempting to close a unit that is being read from or written to
1148	attempting to open a directory
1149	status is 'OLD' and the file is a dangling symbolic link
1150	status is 'NEW' and the file is a symbolic link
1151	no free scratch file names
1152	specifier ACCESS='STREAM' for default unit

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1153	stream-access to default unit
1161	device does not support REWIND
1162	read permission required for BACKSPACE
1163	BACKSPACE on direct-access unit
1164	BACKSPACE on binary unit
1165	end-of-file seen while backspacing
1166	write permission required for ENDFILE
1167	ENDFILE on direct-access unit
1168	stream-access to sequential or direct-access unit
1169	stream-access to unconnected unit
1170	direct-access to stream-access unit
1171	incorrect value of POS specifier
1172	unassociated ASYNCHRONOUS specifier
1173	unassociated DECIMAL specifier
1174	unassociated IOMSG specifier
1175	unassociated ROUND specifier
1176	unassociated STREAM specifier
1177	ASYNCHRONOUS specifier is not 'YES' or 'NO'
1178	ROUND specifier is not 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE' or 'PROCESSOR-DEFINED'
1179	DECIMAL specifier is not 'POINT' or 'COMMA'
1180	RECL specifier is not allowed in OPEN statement for stream-access unit
1181	attempting to allocate an allocated array
1182	deallocating an unassociated pointer
1183	deallocating an unallocated allocatable array
1184	deallocating an allocatable array through a pointer
1185	deallocating an object not allocated by an ALLOCATE statement
1186	deallocating a part of an object
1187	deallocating a larger object than was allocated

TABLE A-2 f95 Runtime I/O Messages *(Continued)*

Error	Message
1191	unallocated array passed to array intrinsic function
1192	illegal rank
1193	small source size
1194	zero array size
1195	negative elements in shape
1196	illegal kind
1197	nonconformable array
2001	invalid constant, structure, or component name
2002	handle not created
2003	character argument too short
2004	array argument too long or too short
2005	end of file, record, or directory stream

APPENDIX **B**

Features Release History

This Appendix lists the new and changed features in this and previous release of `f77` and `f95`:

Fortran 95 New Features and Changes

This section lists the new features and behavior changes introduced with this release of `f95` and previous releases.

`f95` New Features in Sun WorkShop 6 update 2:

The following lists new and changed features in the Fortran 95 compiler released with Sun WorkShop 6 update 2:

- `ALLOCATABLE` Attribute Extended: Recent decisions by the Fortran 95 standards organizations have extended the data entities allowed for the `ALLOCATABLE` attribute. Previously this attribute was limited to locally stored array variables. It is now allowed on:
  - array components of structures
  - dummy arrays
  - array function results

  Allocatable entities remain forbidden in all places where they may be storage-associated (`COMMON` and `EQUIVALENCE` statements). Allocatable array components may appear in `SEQUENCE` types, but objects of such types are then prohibited from `COMMON` and `EQUIVALENCE`. See Appendix C, page 157.

- VALUE Attribute from Fortran 2000: f95 recognizes the VALUE type declaration attribute. Specifying a subprogram dummy input argument with this attribute indicates that the actual argument is passed "by value". See Appendix C, page 157.
- OpenMP 2.0 Fortran API Supported: f95 now supports the OpenMP 2.0 API specifications for Fortran 95. Enhancements include WORKSHARE, REDUCTION for arrays, THREADPRIVATE for variables, COPYPRIVATE for SINGLE directives.

  See http://www.openmp.org/specs for the OpenMP 2.0 specifications. See also Appendix E, page 180.
- OpenMP Library Interface: The compiler now provides an include file 'omp_lib.h' and an interface module omp_lib for defining the interfaces to the OpenMP Fortran library routines. See Appendix E, page 187.
- Interprocedural Optimization (-xipo): This new compiler flag performs whole-program optimizations by invoking an interprocedural analysis pass. Unlike -xcrossfile, -xipo will perform optimizations across all object files in 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. See Chapter 3, page 107.
- VAX Fortran Structures: To aid migration of programs from f77, f95 accepts VAX Fortran STRUCTURE and UNION statements, a precursor of Fortran 95 "derived types". See Appendix C, page 158.
- Stream I/O: Another feature proposed for Fortran 2000 is a new "stream I/O" scheme, which treats a data file as a continuous sequence of bytes, addressable by a positive integer starting from 1. Enable stream I/O by declaring a file with ACCESS='STREAM'. Position files with READ or WRITE statements with the POS=*integer_expression* specifier. See Appendix C, page 158.
- Global Program Checking: Invoked by the -Xlist options, GPC on f95 now looks more like f77, and includes suboptions -Xlistc -Xlist -Xlists -Xlistv*n* and -Xlistw[*n*]. See Chapter 3, page 93.
- Fortran Library Interface: f95 recognizes the include file system.inc for declaring the proper data types for the Fortran library. Supply the statement INCLUDE 'system.inc' in every routine that, references non-intrinsic Fortran library routines to insure proper typing of return values. See the *Fortran Library Reference*.

f95 New Features in Sun WorkShop 6 update 1:

The following lists new and changed features in the Fortran 95 compiler released with Sun WorkShop 6 update 1:

- UltraSPARC III Support: The `-xtarget` and `-xchip` options now accept `ultra3`, and the compiler will generate optimized code for the UltraSPARC III processor. See Chapter 3, page 120.
- Prefetch added to `-fast`: The `-xprefetch` flag has been added to the `-fast` option set. `-fast` automatically sets a number of optimization flags for best execution speed on the compiling platform. Adding `-xprefetch` takes advantage of the UltraSPARC II and III prefetch mechanism, and can add a substantial performance gain in code with loops that process data, See Chapter 3, page 59.
- Support for the `int2` Intrinsic: The Fortran 95 (and Fortran 77) compilers now support the `int2` intrinsic for conversion of data types to 2-byte integer. Use of `int2` as an intrinsic (`M=int2(J)`) appears in many legacy Fortran 77 codes, and is implemented in the Fortran 95 compiler for compatibility. `int` is the preferred Fortran 95 standard intrinsic (`M=int(J,2)`).
- Mixed-Language Linking with `-xlang`: The new `-xlang` option provides an easy way to link object files and libraries compiled by `f77` with `f95` object files. The proper runtime environment is insured when using `-xlang`. See the `CC`(1) man page, and Chapter 3, page 109.

`f95` New Features in Sun WorkShop 6:

The following lists the new and changed features in the Fortran 95 compiler released with Sun Performance WorkShop 6:

- Compliance: The `f95` is fully compliant with the Fortran 95 standard.
- New Command: The Fortran 95 compiler can be invoked by either the `f95` or `f90` command.
- Debugging Optimized Code: Restrictions limiting use of `-g` with other options has been relaxed, allowing debugging parallelized and `-O4` or `-O5` optimized codes with dbx and the Sun WorkShop debugger.
- Source Filename Extensions: The compiler will accept source files with `.f95` and `.f90` filename extensions as well as `.F95` and `.F90`.
- Interval Arithmetic: This release implements a number of extensions that enable interval arithmetic computations. See the *Interval Arithmetic Programming Reference*, and `interval_arithmetic` README for details.
- Enhanced Array Optimizations: The compiler now performs aggressive array optimizations at levels `-O4` and `-O5`.
- Hyper-Linked Diagnostic Messages: Sun WorkShop online help now interprets `f95` error diagnostics in the Building window, creating hypertext links from the error message to descriptive online help.
- OpenMP: The compiler accepts OpenMP explicit parallelization directives. The OpenMP specifications can be viewed at `http://www.openmp.org/`

- AUTOSCOPE added to Cray-style DOALL parallelization directive.
- New/Changed Command-Line Options:
  - -aligncommon aligns COMMON block elements to specific byte boundaries.
  - -r8const promotes single-precision data constants to REAL*8.
  - -xinterval and -xia enable interval arithmetic extensions.
  - -xmemalign specifies general alignment in memory of data elements.
  - -mp=openmp and -openmp enable native compilation of OpenMP explicit parallelization directives.
  - -xprefetch (for enabling UltraSPARC prefetch instructions) has been expanded to include additional sub-options.
  - -xrecursive allows recursive calls from subprograms without the RECURSIVE attribute.
  - -xtypemap has an expanded set of possible data type specifications.
  - -fast extended to set -O5, -fsimple=2, -xvector=yes, and -pad=common.
- Use of f95's parallelization features requires a Sun WorkShop HPC license.

New Features Released In f90 2.0:

The following new and changed features appeared in the f90 2.0 compiler released with Sun WorkShop 5.0 over the earlier f90 1.2 release:

- New options:
  - Most f77 options now recognized by f90.
  - -fpover detects floating-point overflows in I/O processing.
  - -xcode=*code* specifies the memory address model on SPARC platforms.
  - -xcommonchk enables runtime checking for inconsistent COMMON block declarations.
  - -xprefetch allows the compiler to generate prefetch instructions on UltraSPARC II platforms.
  - -xvector allows the compiler to replace certain math library calls within DO loops with single calls to a vectorized math routine.
- Changed options:
  - -xcrossfile[=*n*] – optional level number added.
  - -fns[={yes|no}] – optional yes/no added.
  - -Ztha – option now ignored.
- New Features:
  - Compile for the 64-bit Solaris 7 environment on 64-bit SPARC platforms with -xarch=v9 or v9a.
  - Support in the I/O library for large files (larger than 2 Gigabytes).
  - Support for large arrays on 64-bit Solaris operating environments.
  - Accepts Sun-style directives by default.
  - The REDUCTION directive accepts arrays in the list of variables.
  - *SPARC:* A TASKCOMMON directive declares variables in COMMON to be private.

- New optimization pragma allows setting the compilers optimization level on a routine by routine basis.

- I/O Differences (Comparing `f90` 2.0 against the 1.2 release):
  - NAMELIST Output Format:

    1.2: All variables in a single print statement written to a single line without line breaks. *2.0:* Each variable printed to a separate line.

    1.2: Comma used to separate values. *2.0:* Single blank separates values.

    1.2: Repeated values output using the r* form: 3*8.22 *2.0:* All repeated values output explicitly: 8.22 8.22 8.22

    1.2: No trailing zero printing integer floating point: 1. *2.0:* Floating point integers print with trailing zero: 1.0

    1.2: Value printed may not be the same value when read into a variable with the same type: 0.1 when read in will print as 0.100000001 *2.0:* Prints the minimum number of digits required to ensure that a value written produces the same value when read back in: 0.1 prints as 0.1

    1.2: As required by the standard, zero value prints in exponent form. But 1.2 prints 0.E+0 *2.0:* Prints zero as 0.0E+0

    1.2: Prints a space between the comma and the imaginary part of a complex value: (1., 0.E+0) *2.0:* No comma: (1.0,0.0E+0)

  - NAMELIST Input Format:

    2.0: Allow the group name to be preceded by `$` or `&` on input. The `&` is the only form accepted by the Fortran 90 standard, and is what is written by NAMELIST output.

    2.0: Accepts `$` as the symbol terminating input except if the last data item in the group is CHARACTER, in which case it is treated as input data.

    2.0: Allows NAMELIST input to start in the first column of a record.

  - `PRINT *` no longer comma-delimits output.
  - `OPEN FORM='BINARY'` permits I/O of non-standard raw text without record marks: 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. See Appendix C or the *Fortran 77 Language Reference* for details.
  - Recursive I/O possible on different units (this is because the `f90` I/O library is "MT-Warm").

- RECL=2147483646 (2^{31}-2) is the default record length on sequential formatted, list directed, and namelist output. (Default was 267).
- `ENCODE` and `DECODE` are recognized and implemented as described in the *FORTRAN 77 Language Reference Manual.*
- Naming of scratch files is the same as with `f77`.
- Non-advancing I/O is enabled with `ADVANCE='NO'`, as in:

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

- Handling of I/O on internal files follows the Fortran 90 standard more closely than was the case with `f90` 1.2. Also, calls to routines that do internal I/O are allowed on I/O lists. This was not allowed with 1.2 (or `f77`).

- Operational Differences:
  - Modules are handled differently: Compiling a source code that contains one or more `MODULE` units now causes an information file (name.mod) to be generated for each module. The name of this information file is the name of the module, in lower case, with `.mod` suffix. A `.mod` file must be available before the module can appear on a `USE` statement. This means that all `MODULE` files must be compiled (and the module information files created) before compiling any file referencing a `MODULE` in a `USE` statement
  - `-ftrap=common` is the default trapping mode.
  - Routines from the Sun Performance Library are automatically linked to perform array operations.
- New Language Elements:
  - Some Fortran 95 elements are implemented:
    The attributes `PURE` and `ELEMENTAL`
    The enhanced forms of `MAXVAL` and `MINVAL`
  - New data types are recognized:
    `COMPLEX*32 REAL*16`
    `INTEGER*8` (also `*1`, `*2`)
    `LOGICAL*8` (also `*1`, `*2`)
  - Some data representations have changed from `f90` 1.2:
    `INTEGER*2` is now 2 bytes, not 4
    `INTEGER*1` is now 1 byte, not 4
    `LOGICAL*2` is now 2 bytes, not 4
    `LOGICAL*1` is now 1 byte, not 4
    This will affect programs that read binary data files containing these data items that were written with `f90` programs compiled with the 1.2 compiler. A workaround would be to change the declarations to be `INTEGER*4` or `LOGICAL*4` instead of `*1` or `*2` when compiling with 2.0.

- Call by value, `%VAL`, is implemented in the same manner as `f77`. The only difference is that `f90` 2.0 allows `REAL*8` and `REAL*16` to be passed to C routines as doubles and long doubles.

- `f77` and C Interoperability with `f90` 2.0:
  - To mix `f77` and `f90` object binaries, link with the `f77` compatibility library, `libf77compat`, and not with `libF77`. For example, perform the link step with **`f90`** *..files..* **`-lf77compat`** even if the main program is an `f77` program.
  - The structure of `f90` COMMON is now compatible with `f77`.
  - `f90` *scalar* pointers are compatible with C pointers.

Fortran 77 New Features and Changes

This section lists the new features and behavior changes in `f77` introduced with this and previous releases.

`f77` New Features in Sun WorkShop 6 update 2:

No new features were introduced in `f77` with the release of Sun WorkShop 6 update 2.

`f77` New Features in Sun WorkShop 6 update 1:

The release of Sun WorkShop 6 update 1 introduced the following new or changed features in `f77`:

- UltraSPARC III Support: The `-xtarget` and `-xchip` options now accept `ultra3`, and the compiler will generate optimized code for the UltraSPARC III processor. See Chapter 3, page 120.
- Prefetch added to `-fast`: The `-xprefetch` flag has been added to the `-fast` option set. `-fast` automatically sets a number of optimization flags for best execution speed on the compiling platform. Adding `-xprefetch` takes advantage of the UltraSPARC II and III prefetch mechanism, and can add a substantial performance gain in code with loops that process data, See Chapter 3, page 59.

f77 New Features in Sun WorkShop 6:

Sun WorkShop 6 Fortran 77 includes the following new and changed features:

- I/O Extension: Opening a file with `OPEN(FORM='BINARY')` treats the file as a sequential binary (unformatted) file with no record marks. See the *Fortran 77 Language Reference* for details.
- Debugging Optimized Code: Restrictions limiting use of `-g` with other options has been relaxed, allowing debugging parallelized and `-O4` or `-O5` optimized codes with dbx and the Sun WorkShop debugger.
- New/Changed Command-Line Options:
  - `-aligncommon` aligns COMMON block elements to specific byte boundaries.
  - `-r8const` promotes single-precision data constants to REAL*8
  - `-xmemalign` specifies general alignment in memory of data elements.
  - `-xprefetch` (for enabling UltraSPARC prefetch instructions) has been expanded to include additional sub-options.
  - `-xtypemap` has an expanded set of possible data type specifications.
  - `-fast` extended to set `-O5`, `-fsimple=2`, `-xvector=yes`, and `-pad=common`.
- Use of `f77`'s parallelization features requires a Sun WorkShop HPC license.
- Hyper-Linked Diagnostic Messages: Sun WorkShop online help now interprets `f77` error diagnostics in the Building window, creating hypertext links from the error message to descriptive online help.

Features in f77 5.0:

`f77` 5.0 included the following new and changed features:

- New options:
  - `-fpover` detects floating-point overflows in I/O processing.
  - `-xcode=`*code* specifies the memory address model on SPARC platforms.
  - `-xcommonchk` enables runtime checking for inconsistent COMMON block declarations.
  - `-xmaxopt` enables the `OPT=`*n* pragma and controls the maximum optimization level allowed by `OPT` pragmas in the source code.
  - `-xprefetch` allows the compiler to generate prefetch instructions on UltraSPARC II platforms.
  - `-xvector` allows the compiler to replace certain math library calls within DO loops with single calls to a vectorized math routine.
- Changed options:
  - `-xcrossfile[=`*n*`]` – optional level number added.
  - `-fns[={yes|no}]` – optional yes/no added.
  - `-Ztha` – option now ignored.
- New Features:

- Compile for the 64-bit Solaris 7 environment on 64-bit SPARC platforms with `-xarch=v9` or `v9a`.
- Support in the I/O library for large files (larger than 2 Gigabytes).
- Support for large arrays on 64-bit Solaris 7 environments.
- Dynamic arrays (local arrays with dynamic size) implemented (see *FORTRAN 77 Language Reference Manual*).
- The `REDUCTION` directive accepts arrays in the list of variables.
- *SPARC:* A `TASKCOMMON` directive declares variables in COMMON to be private.
- Fortran 90 style constants that allows specification of byte size (for example, 12345678_8 for a 64-bit, 8-byte, constant).
- New optimization pragma allows setting the compilers optimization level on a routine by routine basis.
- Year 2000 safe `date_and_time()` library routine.

Features in `f77` 4.2:

`f77` 4.2 included the following features that were new or changed since the 4.0 release:

- New options:
  - `-dbl_align_all`
  - `-errtags=yes|no` and `-erroff=`*taglist*
  - `-stop_status=no|yes`
  - `-xcrossfile`
  - `-xlic_lib=`*libs*
  - `-xpp=fpp|cpp`
  - `-xtypemap=`*type:spec,.*
- Changed options:
  - Options `-fround, -fsimple, -ftrap, -xprofile=tcov, -xspace, -xunroll` now available on Intel platforms.
  - `-xtarget, -xarch, -xchip` expanded for SPARC Ultra and Intel platforms.
  - `-vax=` expanded to enable selection/deselection of individual VAX/VMS Fortran features.
  - Default sourcefile preprocessor is *fpp*(1) rather than *cpp*(1).

FORTRAN 77 Upward Compatibility

The FORTRAN 77 5.0 *source* is compatible with earlier releases, except for minor changes due to operating system changes and bug fixes.

Fortran 3.0/3.0.1 to 4.0

Executables (`a.out`), libraries (`.a`), and object files (`.o`) compiled and linked in Fortran 3.0/3.0.1 under Solaris 2 are compatible with Fortran 5.0 under Solaris 2.

BCP: Running Applications from Solaris 1

You must install the Binary Compatibility Package for the executable to run.

Executables compiled and linked in Solaris 1 do run in Solaris 2, but they do not run as fast as when they are compiled and linked under the appropriate Solaris release.

Libraries (`.a`) and object files (`.o`) compiled and linked in Fortran 2.0.1 under Solaris 1 are *not* compatible with Fortran 5.0.

APPENDIX C

Fortran 95 Features and Differences

This appendix shows some of the major features differences between:

- Standard Fortran 95 and Sun Fortran 95
- FORTRAN 77 and Fortran 95

Features and Extensions

Sun WorkShop 6 Fortran 95 provides the following features.

Continuation Line Limits

`f95` and `f77` allow 99 continuation lines (1 initial and 99 continuation lines). Standard Fortran 95 allows 19 for fixed-form and 39 for free-form.

Fixed-Form Source Lines

In fixed-form source, lines can be longer than 72 characters, but everything beyond column 73 is ignored. Standard Fortran 95 only allows 72-character lines.

Directives

`f95` allows directive lines starting with `CDIR$`, `!DIR$`, `CMIC$`, `C$PRAGMA`, or `C$OMP`, in fixed format, or `!DIR$`, `!MIC$`, `!$PRAGMA`, or `!$OMP` in either fixed or free format. For a summary of directives, see Appendix E. Standard Fortran 95 does not consider directives.

- Tabs in `f95` force the rest of the line to be padded out to column 72. This may cause unexpected results if the tab appears within a character string that is continued onto the next line:

Source file:

```
^Iprint *, "Tab on next line
^I1this  continuation line starts with a tab."
^Iend
```

Running the code:

```
Tab on next line                                        this  continuation
 line starts with a tab.
```

Source Form Assumed

The source form assumed by `f95` depends on options, directives, and suffixes.

Files with a `.f` or `.F` suffix are assumed to be in fixed format. Files with a `.f90`, `.f95`, `.F90`, or `.F95` suffix are assumed to be in free format.

TABLE C-1 F95 Source Form Command-line options

Option	Action
`-fixed`	Interpret all source files as Fortran *fixed* form
`-free`	Interpret all source files as Fortran *free* form

If the `-free` or `-fixed` option is used, it overrides the file name suffix. If either a `!DIR$ FREE` or `!DIR$ FIXED` directive is used, it overrides the option and file name suffix.

Mixing Forms

Some mixing of source forms is allowed.

- In the same `f95` command, some source files can be fixed form, some free.
- In the same file, free form *can* be mixed with fixed form by using `!DIR$ FREE` and `!DIR$ FIXED` directives.

Case

Sun Fortran 95 is case insensitive by default. That means that a variable `AbcDeF` is treated as if it were spelled `abcdef`. Compile with the `-U` option to have the compiler treat upper and lower case as unique.

Known Limits

A single Fortran 95 program unit can define up to 65,535 derived types and 16,777,215 distinct constants.

Boolean Type

`f95` supports constants and expressions of Boolean type. There are no Boolean variables or arrays, and there is no Boolean type statement.

Miscellaneous Rules Governing Boolean Type

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

Alternate Forms of Boolean Constants

`f95` allows a Boolean constant (octal, hexadecimal, or Hollerith) in the following alternate forms (no binary). Variables cannot be declared Boolean. Standard Fortran does not allow these forms.

Octal

dddddd`B`, where *d* is any octal digit

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

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

Hexadecimal

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

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

Hollerith

Accepted forms for Hollerith data are:

*n*H...	'...'H	"..."H
*n*L...	'...'L	"..."L
*n*R...	'...'R	"..."R

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

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

Examples: Octal and hexadecimal constants.

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

Examples: Octal and hexadecimal in assignment statements.

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

Use of an octal or hexadecimal constant in an arithmetic expression can produce undefined results and do not generate syntax errors.

Alternate Contexts of Boolean Constants

`f95` allows BOZ constants in the places other than `DATA` statements.

`B'`*bbb*`'` `O'`*ooo*`'` `Z'`*zzz*`'`

`B"`*bbb*`"` `O"`*ooo*`"` `Z"`*zzz*`"`

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

Standard Fortran allows these only in `DATA` statements.

Abbreviated Size Notation for Numeric Data Types

`f95` allows the following nonstandard type declaration forms in declaration statements, function statements, and `IMPLICIT` statements. The form in column one is nonstandard Fortran 95, though in common use. The kind numbers in column two can vary by vendor.

TABLE C-2 Size Notation for Numeric Data Types

Nonstandard	Declarator	Short Form	Meaning
`INTEGER*1`	`INTEGER(KIND=1)`	`INTEGER(1)`	One-byte signed integers
`INTEGER*2`	`INTEGER(KIND=2)`	`INTEGER(2)`	Two-byte signed integers
`INTEGER*4`	`INTEGER(KIND=4)`	`INTEGER(4)`	Four-byte signed integers
`LOGICAL*1`	`LOGICAL(KIND=1)`	`LOGICAL(1)`	One-byte logicals
`LOGICAL*2`	`LOGICAL(KIND=2)`	`LOGICAL(2)`	Two-byte logicals
`LOGICAL*4`	`LOGICAL(KIND=4)`	`LOGICAL(4)`	Four-byte logicals
`REAL*4`	`REAL(KIND=4)`	`REAL(4)`	IEEE single-precision floating-point (Four-byte)
`REAL*8`	`REAL(KIND=8)`	`REAL(8)`	IEEE double-precision floating-point (Eight-byte)
`REAL*16`	`REAL(KIND=16)`	`REAL(16)`	IEEE quad-precision floating-point (Sixteen-byte)

TABLE C-2 Size Notation for Numeric Data Types *(Continued)*

Nonstandard	Declarator	Short Form	Meaning
`COMPLEX*8`	`COMPLEX(KIND=4)`	`COMPLEX(4)`	Single-precision complex (Four-bytes each part)
`COMPLEX*16`	`COMPLEX(KIND=8)`	`COMPLEX(8)`	Double-precision complex (Eight-bytes each part)
`COMPLEX*32`	`COMPLEX(KIND=16)`	`COMPLEX(16)`	Quad-precision complex (Sixteen-bytes each part)

Cray Pointers

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

`f95` supports Cray pointers; Standard Fortran 95 does not.

Syntax

The Cray `POINTER` statement has the following format:

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

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

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

Example: Declare Cray pointers to two pointees.

```
    POINTER ( p, b ),  ( q, c )
```

The above example declares Cray pointer `p` and its pointee `b`, and Cray pointer `q` and its pointee `c`.

Example: Declare a Cray pointer to an array.

```
POINTER ( ix, x(n, 0:m) )
```

The above example declares Cray pointer `ix` and its pointee `x`; and declares `x` to be an array of dimensions `n` by `m+1`.

Purpose of Cray Pointers

You can use pointers to access user-managed storage by dynamically associating variables to particular locations in a block of storage.

Cray pointers allow accessing absolute memory locations.

Cray Pointers and Fortran 95 Pointers

Cray pointers are declared as follows:

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

Fortran 95 pointers are declared as follows:

`POINTER` *object_name*

The two kinds of pointers cannot be mixed.

Features of Cray Pointers

- Whenever the pointee is referenced, `f95` uses the current value of the pointer as the address of the pointee.
- The Cray pointer type statement declares both the pointer and the pointee.
- The Cray pointer is of type Cray pointer.
- The value of a Cray pointer occupies one storage unit on 32-bit processors, and two storage units on 64-bit SPARC V9 processors.
- The Cray pointer can appear in a `COMMON` list or as a dummy argument.
- The Cray pointee has no address until the value of the Cray pointer is defined.
- If an array is named as a pointee, it is called a *pointee array*.

Its array declarator can appear in:

- A separate type statement
- A separate `DIMENSION` statement
- The pointer statement itself

- If the array declarator is in a subprogram, the dimensioning can refer to:
  - Variables in a common block, or
  - Variables that are dummy arguments
- The size of each dimension is evaluated on entrance to the subprogram, not when the pointee is referenced.

Restrictions on Cray Pointers

- *pointee_name* must not be a variable typed `CHARACTER*(*)`.
- If *pointee_name* is an array declarator, it must be explicit shape, (constant or non-constant bounds), or assumed-size.
- An array of Cray pointers is not allowed.
- A Cray pointer cannot be:
  - Pointed to by another Cray pointer or by a Fortran pointer.
  - A component of a structure.
  - Declared to be any other data type.
- A Cray pointer cannot appear in:
  - A `PARAMETER` statement or in a type declaration statement that includes the `PARAMETER` attribute.
  - A `DATA` statement.

Restrictions on Cray Pointees

- A Cray pointee cannot appear in a `SAVE`, `DATA`, `EQUIVALENCE`, `COMMON`, or `PARAMETER` statement.
- A Cray pointee cannot be a dummy argument.
- A Cray pointee cannot be a function value.
- A Cray pointee cannot be a structure or a structure component.
- A Cray pointee cannot be of a derived type.

Usage of Cray Pointers

Cray pointers can be assigned values as follows:

- Set to an absolute address

Example: `q = 0`

- Assigned to or from integer variables, plus or minus expressions

Example: `p = q + 100`

- Cray pointers are not integers. You cannot assign them to a real variable.
- The `LOC` function (nonstandard) can be used to define a Cray pointer.

Example: `p = LOC( x )`

Example: Use Cray pointers as described above.

```
SUBROUTINE  sub ( n )
COMMON pool(100000)
INTEGER blk(128), word64
REAL a(1000), b(n), c(100000-n-1000)
POINTER ( pblk, blk ), (ia, a ), ( ib, b ), &
    ( ic, c ), ( address, word64 )
DATA address / 64 /
pblk = 0
ia = LOC( pool )
ib = ia + 4000
ic = ib + n
...
```

Remarks about the above example:

- `word64` refers to the contents of absolute address 64
- `blk` is an array that occupies the first 128 words of memory
- `a` is an array of length 1000 located in blank common
- `b` follows `a` and is of length `n`
- `c` follows `b`
- `a`, `b`, and `c` are associated with `pool`
- `word64` is the same as `blk(17)` because Cray pointers are byte address and the integer elements of `blk` are each 4 bytes long

Other Language Extensions

Extended ALLOCATABLE Attribute

Recent decisions by the Fortran 95 standards organizations have extended the data entities allowed for the ALLOCATABLE attribute. Previously this attribute was limited to locally stored array variables. It is now allowed with:

- array components of structures
- dummy arrays
- array function results

Allocatable entities remain forbidden in all places where they may be storage-associated: COMMON blocks and EQUIVALENCE statements. Allocatable array components may appear in SEQUENCE types, but objects of such types are then prohibited from COMMON and EQUIVALENCE.

VALUE Attribute (Fortran 2000)

The f95 compiler recognizes the VALUE type declaration attribute. This attribute has been proposed for the Fortran 2000 standard.

Specifying a subprogram dummy input argument with this attribute indicates that the actual argument is passed "by value". The following example demonstrates the use of the VALUE attribute with a C main program calling a Fortran 95 subprogram with a literal value as an argument:

```
C code:
#include <stdlib.h>
int main(int ac, char *av[])
{
    to_fortran(2);
}

Fortran code:
       subroutine to_fortran(i)
       integer, value :: i
       print *, i
       end
```

Stream I/O (Fortran 2000)

A new "stream" I/O scheme has been proposed as part of the Fortran 2000 draft standard. 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*.

`STRUCTURE` and `UNION` (VAX Fortran)

To aid the migration of programs from `f77`, `f95` accepts VAX Fortran `STRUCTURE` and `UNION` statements, a precursor to the "derived types" in Fortran 95. For syntax details see the *FORTRAN 77 Language Reference* manual.

The field declarations within a `STRUCTURE` can be one of the following:

- A substructure — either another `STRUCTURE` declaration, or a record that has been previously defined.
- A `UNION` declaration.
- A `TYPE` declaration, which can include initial values.
- A derived type having the `SEQUENCE` attribute. (This is particular to f95 only.)

As with `f77`, a `POINTER` statement cannot be used as a field declaration.

`f95` also allows:

- Either '`.`' or '`%`' can be used as a structure field dereference symbol: `struct.field` or `struct%field`.
- Structures can appear in a formatted I/O statement.
- Structures can be initialized in a `PARAMETER` statement; the format is the same as a derived type initialization.
- Structures can appear as components in a derived type, but the derived type must be declared with the `SEQUENCE` attribute.

I/O Extensions

Some I/O extensions that appear in Sun Fortran 77 have been added to the Fortran 95 compiler:

- NAMELIST Input Format:

  The group name may be preceded by `$` or `&` on input. The `&` is the only form accepted by the Fortran 95 standard, and is what is written by NAMELIST output.

Accepts $ as the symbol terminating input except if the last data item in the group is CHARACTER data, in which case the $ is treated as input data.

Allows NAMELIST input to start in the first column of a record.

- OPEN(...,FORM='BINARY') treats the file as binary data without record marks:

  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.
- Recursive I/O possible on different units (this is because the f95 I/O library is "MT-Warm").
- RECL=2147483646 (2^{31}-2) is the default record length on sequential formatted, list directed, and namelist output.
- ENCODE and DECODE are recognized and implemented as described in the *FORTRAN 77 Language Reference Manual*.
- Naming of scratch files is the same as with f77.
- Non-advancing I/O is enabled with ADVANCE='NO', as in:

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

Directives

A compiler *directive* directs the compiler to do some special action. Directives are also called *pragmas.*

A compiler directive is inserted into the source program as one or more lines of text. Each line looks like a comment, but has additional characters that identify it as more than a comment for this compiler. For most other compilers, it is treated as a comment, so there is some code portability.

Sun-style directives are the default with `f95` (and `f77`). To switch to Cray-style directives, use the `-mp=cray` compiler command-line flag.

A complete summary of Fortran directives appears in Appendix E.

Form of Special `f95` Directive Lines

f95 recognizes its own special directives in addition to the general `f95/f77` directives described in Chapter 2. These have the following syntax:

```
!DIR$ d1, d2, …
```

Fixed-Form Source

- Put `CDIR$` or `!DIR$` in columns 1 through 5.
- Directives are listed in columns 7 and beyond.
- Columns beyond 72 are ignored.
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.

Free-Form Source

- Put `!DIR$` followed by a space anywhere in the line.
  The `!DIR$` characters are the first nonblank characters in the line (actually, non-whitespace).
- Directives are listed after the space.
- An *initial* directive line has a blank, tab, or newline in the position immediately after the `!DIR$`.

- A *continuation* directive line has a character other than a blank, tab, or newline in the position immediately after the `!DIR$`.

Thus, `!DIR$` in columns 1 through 5 works for both free-form source and fixed-form source.

`FIXED` and `FREE` Directives

These directives specify the source form of lines following the directive line.

Scope

They apply to the rest of the *file* in which they appear, or until the next `FREE` or `FIXED` directive is encountered.

Uses

- They allow you to switch source forms within a source file.
- They allow you to switch source forms for an `INCLUDE` file. You insert the directive at the start of the `INCLUDE` file. After the `INCLUDE` file has been processed, the source form reverts back to the form being used prior to processing the `INCLUDE` file.

Restrictions

The `FREE/FIXED` directives:

- Each must appear alone on a compiler directive line (not continued).
- Each can appear anywhere in your source code. Other directives must appear within the program unit they affect.

Example: A `FREE` directive.

```
!DIR$ FREE
    DO i = 1, n
    a(i) = b(i) * c(i)
    END DO
```

Parallelization Directives

A *parallelization* directive is a special comment that directs the compiler to attempt to parallelize the next DO loop. These are summarized in Appendix E and described in the Fortran Programming Guide. `f95` recognizes both `f77` Sun and Cray style parallelization directives, as well as the OpenMP Fortran API directives.

Note – *Fortran parallelization features require a Forte HPC license.*

Intrinsics

`f95` supports some intrinsic procedures that are extensions beyond the standard.

TABLE C-3 Nonstandard Intrinsics

Name	Definition	Function Type	Argument Types	Arguments	Notes
`COT`	Cotangent	real	real	`([X=]`*x*`)`	P, E
`DDIM`	Positive difference	double precision	double precision	`([X=]`*x*`,[Y=]`*y*`)`	P, E
`LEADZ`	Get the number of leading 0 bits	integer	Boolean, integer, real, or pointer	`([I=]`*i*`)`	NP, I
`POPCNT`	Get the number of set bits	integer	Boolean, integer, real, or pointer	`([I=]`*i*`)`	NP, I
`POPPAR`	Calculate bit population parity	integer	Boolean, integer, real, or pointer	`([X=]`*x*`)`	NP, I

Notes on the above table:

P	The name can be passed as an argument.
NP	The name cannot be passed as an argument.
E	External code for the intrinsic is called at run time.
I	`f95` generates inline code for the intrinsic procedure.

Compatibility with FORTRAN 77

Standard-conforming, fixed-format (*filename*`.f`) FORTRAN 77 source code is compatible with Fortran 95. Use of non-standard extensions, such as VMS Fortran features, are not compatible and may not compile with `f95`.

Incompatibility Issues Between `f95` and `f77`

The following lists some of the known incompatibility issues that arise when compiling and testing `f77` programs with this release of `f95`. These are due to either missing comparable features in `f95`, or differences in behavior. These items are non-standard extensions to Fortran 77 supported in `f77` but not in `f95`.

- I/O (see also page 158 and page 164):
  - List-directed output uses different formats.
  - Variable format expressions are not available in Fortran 95.
  - You cannot open a file with `ACCESS='APPEND'` in Fortran 95.
  - `f95` does not allow `BACKSPACE` or `ENDFILE` on a direct-access file.
  - Fortran 95 requires explicit field width specifications in format edit descriptors. For example, `FORMAT(I)` is not allowed.
  - `f95` does not recognize `f77` escape sequences (for example, `\n \t \'`) in output formats.
  - `f95` does not recognize `FILEOPT=` in `OPEN` statements.
  - `f95` does not recognize the `'`*n* form for specifying a record number in direct access I/O: `READ (2 '13) X,Y,Z`
- Data Types, Declarations, and Usage:
  - If it appears in a program unit, the `IMPLICIT` statement in Fortran 95 must precede the first declarative statement in the unit.
  - `f95` allows only 7 array subscripts; f77 allows 20.
  - `LOGICAL` and `INTEGER` variables cannot be used interchangeably in Fortran 95.
  - Fortran 95 Cray pointers cannot appear in some intrinsic calls.
  - `f77`-style initializations using slashes on type declarations are not accepted in Fortran 95.
  - Fortran 95 does not allow assigning Cray character pointers to non-pointer variables to other Cray pointers that are not character pointers.
  - Fortran 95 does not allow the same Cray pointer to point to items of different type sizes (for example, `REAL*8` and `INTEGER*4`).
  - Fortran 95 does not accept the `BYTE` data type.
  - Fortran 95 does not allow non-integers to be used as array subscripts.
  - `f95` does not allow relational operators `.EQ.` and `.NE.` to be used with logical operands.

- Programs, Subroutines, Functions, Statements:
  - The `PROGRAM` statement requires a *name* in Fortran 95.
  - The `f95` maximum length for names is 31 characters.
  - Functions in Fortran 95 cannot be called by a `CALL` statement, as if they were subroutines.
  - Functions in Fortran 95 must have their return value defined.
  - While `f77` allows mixed argument types to appear in some specific intrinsic functions, `f95` does not.
  - `f95` does not recognize debugging comments (comment lines with "`D`" in column one).
  - Tab-formatting in Fortran 95 does not allow source lines to extend beyond column 72.
  - `f95` tab-formatting will pad character strings to column 72 if they extend over a continuation line. (See page 148)
- Command-line Options:
  - `f95` does not recognize `-vax` compiler options.

I/O Compatibility

`f77` and `f95` are generally I/O compatible for binary I/O, since `f95` links to the `f77` compatibility library.

Such compatibility includes the following two situations:

- In the same program, you can write some records in `f95`, then read them in `f77`.
- An `f95` program can write a file. Then an `f77` program can read it.

The numbers read back in may or may not equal the numbers written out.

- Unformatted: The numbers read back in do equal the numbers written out.
- Floating-point formatted: The numbers read back in can be different from the numbers written out. This is caused by slightly different base conversion routines, or by different conventions for uppercase/lowercase, spaces, plus or minus signs, and so forth.

  Examples: `1.0e12`, `1.0E12`, `1.0E+12`
- List-directed: The numbers read back in can be different from the numbers written out. This can be caused by various layout conventions with commas, spaces, zeros, repeat factors, and so forth.

  Example: ’0.0’ as compared to ’.0’

  Example: ’ 7’ as compared to ’7’

  Example: ’3, 4, 5’ as compared to ’3 4 5’

Example: '3*0' as compared to '0 0 0'

The above results from: `integer::v(3)=(/0,0,0/); print *,v`

Example: '0.333333343' as compared to '0.333333'

The above results from `PRINT *, 1.0/3.0`

Linking with f77-Compiled Routines

- To mix `f77` and `f95` object binaries, link with `f95` and the `f77` compatibility library, `libf77compat`, and not with `libF77`. The `-xlang=f77` option provides an easy way to do this. Perform the link step with `f95` even if the main program is an `f77` program
- Example: `f95` main and `f77` subroutine.

```
demo% cat m.f95
CHARACTER*74 :: c = 'This is a test.'
   CALL echo1( c )
END
demo$ cat s.f
       SUBROUTINE echo1( a )
       CHARACTER*74 a
       PRINT*, a
       RETURN
       END
demo% f77 -c -silent s.f
demo% f95 -xlang=f77 m.f95 s.o
demo% a.out
 This is a test.
demo%
```

- The FORTRAN 77 library is generally compatible with `f95`.

Example: `f95` main calls a routine from the FORTRAN 77 library.

```
demo% cat tdtime.f95
        REAL e, dtime, t(2)
        e = dtime( t )
        DO i = 1, 100000
           as = as + cos(sqrt(float(i)))
        END DO
        e = dtime( t )
        PRINT *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
        END
demo% f95 tdtime.f95
demo% a.out
elapsed: 0.14 , user: 0.14 , sys: 0.0E+0
demo%
```

See `dtime`(3F).

Intrinsics

The Fortran 95 standard supports the following intrinsic functions that FORTRAN 77 does not have.

If you use one of these names in your program, you must add an `EXTERNAL` statement to make `f95` use your function rather than the intrinsic one.

Fortran 95 intrinsics:

```
ADJUSTL,ADJUSTR,ALL,ALLOCATED,ANY,BIT_SIZE,COUNT,CSHIFT,
DIGITS,DOT_PRODUCT,EOSHIFT,EPSILON,EXPONENT,HUGE,KIND,
LBOUND,LEN_TRIM,MATMUL,MAXEXPONENT,MAXLOC,MAXVAL,MERGE,
MINEXPONENT,MINLOC,MINVAL,NEAREST,PACK,PRECISION,PRESENT,
PRODUCT,RADIX,RANGE,REPEAT,RESHAPE,RRSPACING,SCALE,SCAN,
SELECTED_INT_KIND,SELECTED_REAL_KIND,SET_EXPONENT,SHAPE,
SIZE,SPACING,SPREAD,SUM,TINY,TRANSFER,TRANSPOSE,UBOUND,
UNPACK,VERIFY
```

Forward Compatibility

Future releases of `f95` are intended to be source code compatible with this release.

Module information files generated by this release of `f95` are not guaranteed to be compatible with future releases.

Mixing Languages

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

Module Files

Compiling a file containing a Fortran 95 `MODULE` generates a module interface file (`.mod` file) for every `MODULE` encountered in the source. The file name is derived from the name of the `MODULE`; file `xyz.mod` (all lowercase) will be created for `MODULE xyz`.

Compilation also generates a `.o` module implementation object file for the source file containing the `MODULE` statements. Link with the module implementation object file along with the all other object files to create an executable.

The compiler creates module interface files and implementation object files in the current working directory. It looks in the current working directory for the interface files when compiling `USE` *modulename* statements. The `-M`*path* option allows you to give the compiler an additional path to search. Module implementation object files must be listed explicitly on the command line for the link step.

Typically, programmers define one `MODULE` per file and assign the same name to the `MODULE` and the source file containing it. However, this is not a requirement.

The `.mod` files cannot be stored into an archive file, or concatenated into a single file.

Example:

```
demo% cat mod_one.f90
 MODULE one
    ...
 END MODULE
demo% cat mod_two.f90
 MODULE two
    ...
 END MODULE
demo% cat main.f90
 USE one
 USE two
   ...
 END
demo% f95 -o main mod_one.f90 mod_two.f90 main.f90
```

In this example, all the files are compiled at once. The module source files appear first before their use in the main program. Compilation creates the files:

```
main
main.o
one.mod
mod_one.o
two.mod
mod_two.o
```

The next example compiles each unit separately and links them together.

```
demo% f95 -c mod_one.f90 mod_two.f90
demo% f95 -c main.f90
demo% f95 -o main main.o mod_one.o mod_two.o
```

When compiling `main.f90`, the compiler searches the current directory for `one.mod` and `two.mod`. These must be compiled before compiling any files that reference the modules on `USE` statements. The link step requires the module implementation object files `mod_one.o` and `mod_two.o` appear along with all other object files to create the executable.

APPENDIX **D**

`-xtarget` Platform Expansions

This Appendix details the `-xtarget` option platform system names and their expansions.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options, as shown in the following table. Run `fpversion`(1) to determine the target definitions on any system.

For example:

```
-xtarget=sun4/15
```

means

```
-xarch=v8a -xchip=micro -xcache=2/16/1
```

TABLE D-1 `-xtarget` Expansions

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
generic64	v9	generic	generic
cs6400	v8	super	16/32/4:2048/64/1
entr150	v8plusa	ultra	16/32/1:512/64/1
entr2	v8plusa	ultra	16/32/1:512/64/1
entr2/1170	v8plusa	ultra	16/32/1:512/64/1
entr2/1200	v8plusa	ultra	16/32/1:512/64/1
entr2/2170	v8plusa	ultra	16/32/1:512/64/1
entr2/2200	v8plusa	ultra	16/32/1:512/64/1
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1

TABLE D-1 `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1

TABLE D-1 `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1

TABLE D-1 `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1

TABLE D-1 `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/512/1

APPENDIX **E**

Fortran Directives Summary

This appendix summarizes the directives recognized by the f77 and f95 Fortran compilers:

- General Fortran Directives
- Sun Parallelization Directives
- Cray Parallelization Directives
- OpenMP Fortran 95 Directives, Library Routines, and Environment

Note – *Fortran parallelization features require a Sun WorkShop HPC license.*

General Fortran Directives

General directives accepted by both `f77` and `f95` are described in Chapter 2.

TABLE E-1 Summary of General Fortran Directives

Format	

`C$PRAGMA` *keyword* `(` *a* `[ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…`
`C$PRAGMA SUN` *keyword* `(` *a* `[ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…`
`C$PRAGMA SPARC` *keyword* `(` *a* `[ , a ] … ) [ , keyword ( a [ , a ] … ) ] ,…`
 	
Comment-indicator in column 1 may be `c`, `C`, `!`, or `*`. (We use `C` in these examples. `f95` free-format must use `!`.)	

`C` *Directive*	`C$PRAGMA C(`*list*`)` Declares a list of names of external functions as C language routines.

TABLE E-1 Summary of General Fortran Directives *(Continued)*

`UNROLL` *Directive*	`C$PRAGMA SUN UNROLL=`*n* Advises the compiler that the following loop can be unrolled to a length *n*.
`WEAK` *Directive*	`C$PRAGMA WEAK(`*name*`[=`*name2*`])` Declares *name* to be a weak symbol, or an alias for *name2*.
`OPT` *Directive*	`C$PRAGMA SUN OPT=`*n* Set optimization level for a subprogram to *n*.
`NOMEMDEP` *Directive*	`C$PRAGMA SUN NOMEMDEP` Assert there are no memory dependencies in the following loop. (Requires `-parallel` or `-explicitpar`.)
`PIPELOOP` *Directive*	`C$PRAGMA SUN PIPELOOP=`*n* Assert dependency in loop between iterations *n* apart.

`PREFETCH` *Directives*	`C$PRAGMA SPARC_PREFETCH_READ_ONCE (`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_READ_MANY (`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_WRITE_ONCE (`*name*`)`
	`C$PRAGMA SPARC_PREFETCH_WRITE_MANY (`*name*`)`
	
	Request compiler generate prefetch instructions for references to name. (Requires `-xprefetch` option.)

Special Fortran 95 Directives

The following directives are only available with `f95`. See Appendix C for details.

TABLE E-2 Special Fortran 95 Directives

Format	`!DIR$` *directive* : initial line `!DIR$&` ... : continuation line With fixed-format source, `C` is also accepted as a directive-indicator: `CDIR$` *directive...* ; the line must start in column 1. WIth free-format source, the line may be preceded by blanks.

`FIXED/FREE` *Directives*	`!DIR$ FREE`
	`!DIR$ FIXED`
	
	These directives specify the source format of the lines following the directive. They apply to the rest of the source file in which they appear, up to the next `FREE` or `FIXED` directive.

Sun Parallelization Directives

Sun-style parallelization directives are the default (`-mp=sun` compiler option), and are detailed in the chapter on parallelization in the *Fortran Programming Guide*.

TABLE E-3 Sun-Style Parallelization Directives Summary

Format	`C$PAR` *directive [optional_qualifiers]* : initial line `C$PAR&` *[more_qualifiers]* : continuation line Fixed format, the directive-indicator may be `C` (as shown), `c`, `*`, or `!`. Separate multiple qualifiers with commas. Characters beyond column 72 ignored unless `-e` compiler option specified.
`TASKCOMMON` *Directive*	`C$PAR TASKCOMMON` *block_name* Declares variables in common block *block_name* as thread-private: private to a thread, but global within the thread. Declaring a common block `TASKCOMMON` requires that this directive appear after *every* common declaration of that block.

TABLE E-3 Sun-Style Parallelization Directives Summary *(Continued)*

`DOALL` *Directive*	`C$PAR DOALL` *[qualifiers]* Parallelize DO loop that follows. Qualifiers are: `PRIVATE`(*list*) declare names on list PRIVATE `SHARED`(*list*) declare names on list SHARED `MAXCPUS`(*n*) use no more than *n* threads `READONLY`(*list*) listed variables not modified in loop `SAVELAST` save last value of all private variables `STOREBACK`(*list*) save last value of listed variables `REDUCTION`(*list*) listed variables are reduction variables `SCHEDTYPE`(*type*) use scheduling type: (default is `STATIC`) `STATIC` `SELF`(*nchunk*) `FACTORING`[(*m*)] `GSS`[(*m*)]
`DOSERIAL` *Directive*	`C$PAR DOSERIAL` Disables parallelization of the loop that follows.
`DOSERIAL*` *Directive*	`C$PAR DOSERIAL*` Disables parallelization of the loop nest that follows.

Cray Parallelization Directives

Cray-style parallelization directives are detailed in the chapter on parallelization in the *Fortran Programming Guide*. Requires `-mp=cray` compiler option.

TABLE E-4 Cray Parallelization Directives Summary

Format	`CMIC$` *directive qualifiers* : initial line `CMIC$&` *[more_qualifiers]* : continuation line Fixed format. Directive-indicator may be `C` (as shown here), `c`, `*`, or `!`. With `f95` free-format, leading blanks can appear before `!MIC$`.

`DOALL` *Directive*	`CMIC$ DOALL SHARED`(*list*), `PRIVATE`(*list*) *[, more_qualifiers]*
	
	Parallelize loop that follows. Qualifiers are:
	Scoping qualifiers are required (unless *list* is empty)—all variables in the loop must appear in a `PRIVATE` or `SHARED` clause:
	`PRIVATE`(*list*) declare names on list PRIVATE
	`SHARED`(*list*) declare names on list SHARED
	`AUTOSCOPE` automatically determine scope of variables
	The following are optional:
	`MAXCPUS`(*n*) use no more than *n* threads
	`SAVELAST` save last value of all private variables
	Only one scheduling qualifier may appear:
	`GUIDED` *equivalent to Sun-style* `GSS(64)`
	`SINGLE` *equivalent to Sun-style* `SELF(1)`
	`CHUNKSIZE`(*n*) *equivalent to Sun-style* `SELF(n)`
	`NUMCHUNKS`(*m*) *equivalent to Sun-style* `SELF(n/m)`
	The default scheduling is Sun-style `STATIC`, for which there is no Cray-style equivalent. Interpretations of these scheduling qualifiers differ between Sun and Cray style. Check the *Fortran Programming Guide* for details.

`TASKCOMMON` *Directive*	`CMIC$ TASKCOMMON` *block_name* Declares variables in the named common block as *thread-private*—private to a thread, but global within the thread. Declaring a common block `TASKCOMMON` requires that this directive appear immediately after *every* common declaration of that block.
`DOSERIAL` *Directive*	`CMIC$ DOSERIAL` Disables parallelization of the loop that follows.
`DOSERIAL*` *Directive*	`CMIC$ DOSERIAL*` Disables parallelization of the loop nest that follows.

Fortran 95 OpenMP Directives

The Sun Fortran 95 compiler supports the OpenMP 2.0 Fortran API. The `-openmp` compiler flag enables these directives. (See page 79).

This section lists the OpenMP directives, library routines, and environment variables supported by `f95`. For details about parallel programming with OpenMP, see the OpenMP 2.0 Fortran specification at `http://www.openmp.org/`.

The following table summarizes the OpenMP directives supported by `f95`. Items enclosed in square brackets ([...]) are optional. The compiler permits comments to follow an exclamation mark (`!`) on the same line as the directive. When compiling with `-openmp`, the CPP/FPP variable `_OPENMP` is defined and may be used for conditional compilation within `#ifdef _OPENMP` and `#endif`.

TABLE E-5 Summary of OpenMP Directives in Fortran 95

Directive Format (Fixed)	`C$OMP` *directive optional_clauses...* `!$OMP` *directive optional_clauses...* `*$OMP` *directive optional_clauses...* Must start in column one; continuation lines must have a non-blank or non-zero character in column 6
Directive Format (Free)	`!$OMP` *directive optional_clauses...* May appear anywhere, preceded by whitespace; an ampersand (`&`) at the end of the line identifies a continued line.
Conditional Compilation	Source lines beginning with `!$`, `C$`, or `*$` in columns 1 and 2 (fixed format), or `!$` preceded by white space (free format) are compiled only when compiler option `-openmp`, or `-mp=openmp` is specified.
`PARALLEL` *Directive*	`!$OMP PARALLEL` [*clause*[[,] *clause*]...] *block of Fortran statements with no transfer in or out of block* `!$OMP END PARALLEL` Defines a *parallel region*: a block of code that is to be executed by multiple threads in parallel. *clause* can be one of the following: `PRIVATE`(*list*), `SHARED`(*list*), `DEFAULT`(*option*), `FIRSTPRIVATE`(*list*), `REDUCTION`(*list*), `IF`(*logical_expression*), `COPYIN`(*list*), `NUM_THREADS`(*integer_expression*).

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`DO` *Directive*	`!$OMP DO` [*clause*[[,] *clause*]...] *do_loop statements block* [`!$OMP END DO` [`NOWAIT`]] The `DO` directive specifies that the iterations of the DO loop that immediately follows must be executed in parallel. This directive must appear within a parallel region. *clause* can be one of the following: `PRIVATE`(*list*), `FIRSTPRIVATE`(*list*), `LASTPRIVATE`(*list*), `REDUCTION`(*list*), `SCHEDULE`(*type*), `ORDERED`.
`SECTIONS` *Directive*	`!$OMP SECTIONS` [*clause*[[,] *clause*]...] [`!$OMP SECTION`] *block of Fortran statements with no transfer in or out* [`!$OMP SECTION` *optional block of Fortran statements*] ... `!$OMP END SECTIONS` [`NOWAIT`] Encloses a non-iterative section of code to be divided among threads in the team. Each section is executed once by a thread in the team. *clause* can be one of the following: `PRIVATE`(*list*), `FIRSTPRIVATE`(*list*), `LASTPRIVATE`(*list*), `REDUCTION`(*list*). Each section is preceded by a `SECTION` directive, which is optional for the first section.
`SINGLE` *Directive*	`!$OMP SINGLE` [*clause*[[,] *clause*]...] *block of Fortran statements with no transfer in or out* `!$OMP END SINGLE` [*end-modifier*] The statements enclosed by `SINGLE` is to be executed by only one thread in the team. Threads in the team that are not executing the `SINGLE` block of statements wait at the `END SINGLE` directive unless `NOWAIT` is specified. *clause* can be one of: `PRIVATE`(*list*), `FIRSTPRIVATE`(*list*). *end-modifier* is either `COPYPRIVATE(`*list*`)`[[,]`COPYPRIVATE(`*list*...`)`] or `NOWAIT`.
`WORKSHARE` *Directive*	`!$OMP WORKSHARE` *block of Fortran statements* `!$OMP END WORKSHARE` [`NOWAIT`] Divides the work of executing the enclosed code block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once.

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

PARALLEL DO *Directive*	!$OMP PARALLEL DO [*clause*[[,] *clause*]...] *do_loop statements block* [!$OMP END PARALLEL DO] Shortcut for specifying a parallel region that contains a single DO loop: a PARALLEL directive followed immediately by a DO directive. *clause* can be any of the clauses accepted by the PARALLEL and DO directives.
PARALLEL SECTIONS *Directive*	!$OMP PARALLEL SECTIONS [*clause*[[,] *clause*]...] [!$OMP SECTION] *block of Fortran statements with no transfer in or out* [!$OMP SECTION *optional block of Fortran statements*] ... !$OMP END PARALLEL SECTIONS Shortcut for specifying a parallel region that contains a single SECTIONS directive: a PARALLEL directive followed by a SECTIONS directive. *clause* can be any of the clauses accepted by the PARALLEL and SECTIONS directives.
PARALLEL WORKSHARE *Directive*	!$OMP PARALLEL WORKSHARE[*clause*[[,] *clause*]...] *block of Fortran statements* !$OMP END PARALLEL WORKSHARE Provides a shortcut for specifying a parallel region that contains a single WORKSHARE directive. *clause* can be one of the clauses accepted by either the PARALLEL or WORKSHARE directive.
Synchronization Directives	
MASTER *Directive*	!$OMP MASTER *block of Fortran statements with no transfers in or out* !$OMP END MASTER The block of statements enclosed by these directives is executed only by the master thread of the team. The other threads skip this block and continue. There is no implied barrier on entry to or exit from the master section.

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`CRITICAL` *Directive*	`!$OMP CRITICAL` [(*name*)] *block of Fortran statements with no transfers in or out* `!$OMP END CRITICAL` [(*name*)] Restrict access to the statement block enclosed by these directives to only one thread at a time. The optional *name* argument identifies the critical region. All unnamed `CRITICAL` directives map to the same name. Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is undefined. If *name* appears on the `CRITICAL` directive, it must also appear on the `END CRITICAL` directive.
`BARRIER` *Directive*	`!$OMP BARRIER` Synchronizes all the threads in a team. Each thread waits until all the others in the team have reached this point.
`ATOMIC` *Directive*	`!$OMP ATOMIC` Ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The directive applies only to the immediately following statement, which must be one of these forms: *x = x operator expression* *x = expression operator x* *x = intrinsic(x, expression)* *x = intrinsic(expression, x)* where: • *x* is a scalar of intrinsic type • *expression* is a scalar expression that does not reference *x* • *intrinsic* is one of `MAX, MIN, IAND, IOR,` or `IEOR`. • *operator* is one of `+ - * / .AND. .OR. .EQV. .NEQV.` *This implementation replaces all ATOMIC directives by enclosing the target statement in a critical section.*

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

FLUSH *Directive*	!$OMP FLUSH [(*list*)] Thread-visible variables are written back to memory at the point at which this directive appears. The FLUSH directive only provides consistency between operations within the executing thread and global memory. The optional *list* consists of a comma-separated list of variables that need to be flushed. The FLUSH directive is implied for the following directives: BARRIER, CRITICAL/ENDCRITICAL, ENDDO, END SECTIONS, ENDSINGLE, ENDWORKSHARE, ORDERED/ENDORDERED, PARALLEL/ENDPARALLEL, PARALLEL/ENDPARALLELDO, PARALLELSECTIONS/ENDPARALLELSECTIONS, PARALLELWORKSHARE/ENDPARALLELWORKSHARE. FLUSH is not implied if NOWAIT is specified. It is not implied by: DO, MASTER/ENDMASTER, SECTIONS, SINGLE, and WORKSHARE.
ORDERED *Directive*	!$OMP ORDERED *block of Fortran statements with no transfers in or out* !$OMP END ORDERED The enclosed block of statements are executed in the order that iterations would be executed in a sequential execution of the loop. It can appear only in the dynamic extent of a DO or PARALLEL DO directive. The ORDERED clause must be specified on the closest DO directive enclosing the block.
Data Environment Directives	
THREADPRIVATE *Directive*	!$OMP THREADPRIVATE(*list*) Makes the *list* of variables and named common blocks private to a thread but global within the thread. Common block names must appear between slashes. To make a common block THREADPRIVATE, this directive must appear after every COMMON declaration of that block.
Data Scoping Clauses	
Several directives noted above accept clauses to control the scope attributes of variables enclosed by the directive. If no data scope clause is specified for a directive, the default scope for variables affected by the directive is SHARED. *list* is a comma-separated list of named variables or common blocks that are accessible in the scoping unit. Common block names must appear within slashes (for example, /ABLOCK/)	
PRIVATE *Clause*	PRIVATE(*list*) Declares the variables in the comma separated *list* to be private to each thread in a team.

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`SHARED` *Clause*	`SHARED`(*list*) All the threads in the team share the variables that appear in *list*, and access the same storage area.
`DEFAULT` *Clause*	`DEFAULT(PRIVATE | SHARED | NONE)` Specify scoping attribute for all variables within a parallel region. `THREADPRIVATE` variables are not affected by this clause. If not specified, `DEFAULT(SHARED)` is assumed.
`FIRSTPRIVATE` *Clause*	`FIRSTPRIVATE`(*list*) Variables on list are `PRIVATE`. In addition, private copies of the variables are initialized from the original object existing before the construct.
`LASTPRIVATE` *Clause*	`LASTPRIVATE`(*list*) Variables on the list are `PRIVATE`. In addition, when the `LASTPRIVATE` clause appears on a DO directive, the thread that executes the sequentially last iteration updates the version of the variable before the construct. On a `SECTIONS` directive, the thread that executes the lexically last `SECTION` updates the version of the object it had before the construct.
`REDUCTION` *Clause*	`REDUCTION`([*operator* | *intrinsic*]`:`*list*) *operator* is one of: `+ * - .AND. .OR. .EQV. .NEQV.` *intrinsic* is one of: `MAX MIN IAND IOR IEOR` Variables in *list* must be named variables of intrinsic type. The `REDUCTION` clause is intended to be used on a region in which the reduction variable is used only in reduction statements of the form shown previously for the `ATOMIC` directive. Variables on *list* must be `SHARED` in the enclosing context. A private copy of each variable is created for each thread as if it were `PRIVATE`. At the end of the reduction, the shared variable is updated by combining the original value with the final value of each of the private copies.
`COPYIN` *Clause*	`COPYIN`(*list*) The `COPYIN` clause applies only to variables, common blocks, and variables in common blocks that are declared as `THREADPRIVATE`. In a parallel region, `COPYIN` specifies that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region.

TABLE E-5 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`COPYPRIVATE` *Clause*	`COPYPRIVATE(`*list*`)` Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. Variables in list must not appear in a `PRIVATE` or `FIRSTPRIVATE` clause of the `SINGLE` construct specifying `COPYPRIVATE`..
Scheduling Clauses on `DO` *and* `PARALLEL DO` *Directives*	
`SCHEDULE` *Clause*	`SCHEDULE`(*type* [,*chunk*]) Specifies how iterations of the DO loop are divided among the threads of the team. *type* can be one of the following. In the absence of a `SCHEDULE` clause, `STATIC` scheduling is used.
`STATIC` *Scheduling*	`SCHEDULE(STATIC,` *chunk*`)` Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. *chunk* must be a scalar integer expression.
`DYNAMIC` *Scheduling*	`SCHEDULE(DYNAMIC,` *chunk*`)` Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations.
`GUIDED` *Scheduling*	`SCHEDULE(GUIDED,` *chunk*`)` With `GUIDED`, the *chunk* size is reduced in an exponentially decreasing manner with each dispatched piece of the iterations. *chunk* specifies the minimum number of iterations to dispatch each time. (Default chunk size is 1. The size of the initial piece of the iterations is the number of iterations in the loop divided by the number of threads executing the loop.)
`RUNTIME` *Scheduling*	`SCHEDULE(RUNTIME)` Scheduling is deferred until runtime. Schedule *type* and *chunk* size will be determined from the setting of the `OMP_SCHEDULE` environment variable. (Default is `STATIC`.)

OpenMP Library Routines

OpenMP Fortran API library routines are external procedures. In the following summary, *int_expr* is a default scalar integer expression, and *logical_expr* is a default scalar logical expression.

`OMP_` functions returning `INTEGER(4)` and `LOGICAL(4)` are not intrinsic and must be declared properly, otherwise the compiler will assume `REAL`. Interface declarations for the OpenMP Fortran runtime library routines summarized below are provided by the Fortran include file `omp_lib.h` and a Fortran 95 `MODULE omp_lib`, as described in the Fortran OpenMP 2.0 specifications. Supply an `INCLUDE 'omp_lib.h'` statement or `#include "omp_lib.h"` preprocessor directive, or a `USE omp_lib` statement in every program unit that references these library routines.

Compiling with `-Xlist` will report any type mismatches.

TABLE E-6 Summary of Fortran 95 OpenMP Library Routines

Execution Environment Routines
`OMP_SET_NUM_THREADS` Subroutine `SUBROUTINE OMP_SET_NUM_THREADS(`*int_expr*`)` Sets the number of threads to use for the next parallel region.
`OMP_GET_NUM_THREADS` Function `INTEGER(4) FUNCTION OMP_GET_NUM_THREADS()` Returns the number of threads currently in the team executing the parallel region from which it is called.
`OMP_GET_MAX_THREADS` Function `INTEGER(4) FUNCTION OMP_GET_MAX_THREADS()` Returns the maximum value that can be returned by calls to the `OMP_GET_NUM_THREADS function.`
`OMP_GET_THREAD_NUM` Function `INTEGER(4) FUNCTION OMP_GET_THREAD_NUM()` Returns the thread number within the team. This is a number between 0 and `OMP_GET_NUM_THREADS()-1.` The master thread is thread 0.
`OMP_GET_NUM_PROCS` Function `INTEGER(4) FUNCTION OMP_GET_NUM_PROCS()` Returns the number of processors that are available to the program.

TABLE E-6 Summary of Fortran 95 OpenMP Library Routines *(Continued)*

`OMP_IN_PARALLEL` Function `LOGICAL(4) FUNCTION OMP_IN_PARALLEL()` Returns .`TRUE`. if called from within the dynamic extent of a region executing in parallel, and .`FALSE`. otherwise.
`OMP_SET_DYNAMIC` Subroutine `SUBROUTINE OMP_SET_DYNAMIC`(*logical_expr*) Enables or disables dynamic adjustment of the number of threads available for parallel execution of programs. (Dynamic adjustment is enabled by default).
`OMP_GET_DYNAMIC` Function `LOGICAL(4) FUNCTION OMP_GET_DYNAMIC()` Returns .`TRUE`. if dynamic thread adjustment is enabled and returns .`FALSE`. otherwise.
`OMP_SET_NESTED` Subroutine `SUBROUTINE OMP_SET_NESTED`(*logical_expr*) Enables or disables nested parallelism. (Nested parallelism is disabled by default.) *Nested parallelism is not supported.*
`OMP_GET_NESTED` Function `LOGICAL(4) FUNCTION OMP_GET_NESTED()` Returns .`TRUE`. if nested parallelism is enabled, .`FALSE`. otherwise. *Nested parallelism is not supported; this function will always return* .`FALSE`.

Lock Routines

Two types of locks are supported: simple locks and nestable locks. Nestable locks may be locked multiple times by the same thread before being unlocked; simple locks may not be locked if they are already in a locked state. Simple lock variables may only be passed to simple lock routines, and nested lock variables only to nested lock routines.

The lock variable *var* must be accessed only through these routines. Use the parameters `OMP_LOCK_KIND` and `OMP_NEST_LOCK_KIND` (defined in `omp_lib.h` `INCLUDE` file and the `omp_lib` `MODULE`) for this purpose. For example,

```
INTEGER(KIND=OMP_LOCK_KIND) :: var
INTEGER(KIND=OMP_NEST_LOCK_KIND) :: nvar
```

TABLE E-6 Summary of Fortran 95 OpenMP Library Routines *(Continued)*

`OMP_INIT_LOCK` Subroutine
 	
`SUBROUTINE OMP_INIT_LOCK(`*var*`)`
`SUBROUTINE OMP_INIT_NEST_LOCK(`*nvar*`)`
 	
Initializes a lock associated with lock variable *var* for use in subsequent calls. The initial state is unlocked.

`OMP_DESTROY_LOCK` Subroutine
 	
`SUBROUTINE OMP_DESTROY_LOCK(`*var*`)`
`SUBROUTINE OMP_DESTROY_NEST_LOCK(`*nvar*`)`
 	
Disassociates the given lock variable *var* from any locks.

`OMP_SET_LOCK` Subroutine
 	
`SUBROUTINE OMP_SET_LOCK(`*var*`)`
`SUBROUTINE OMP_SET_NEST_LOCK(`*nvar*`)`
Forces the executing thread to wait until the specified lock is available. The thread is granted ownership of the lock when it is available.

`OMP_UNSET_LOCK` Subroutine
 	
`SUBROUTINE OMP_UNSET_LOCK(`*var*`)`
`SUBROUTINE OMP_UNSET_NEST_LOCK(`*nvar*`)`
Releases the executing thread from ownership of the lock. Behavior is undefined if the thread does not own that lock.

`OMP_TEST_LOCK` Function
 	
`LOGICAL FUNCTION OMP_TEST_LOCK(`*var*`)`
`INTEGER FUNCTION OMP_TEST_NEST_LOCK(`*nvar*`)`
Attempts to set the lock associated with lock variable. Returns `.TRUE.` if the simple lock was set successfully, `.FALSE.` otherwise. `OMP_TEST_NEST_LOCK` returns the new nesting count if the lock associated with *nvar* was set successfully, otherwise it returns 0.

Timing Routines These two functions, returning double precision (`REAL(8)`), support a portable wall-clock timer.
`OMP_GET_WTIME` *Function* `REAL(8) FUNCTION OMP_GET_WTIME()` Returns a double precision value equal to the elapsed wallclock time in seconds since "some arbitrary time in the past"
`OMP_GET_WTICK` *Function* `REAL(8) FUNCTION OMP_GET_WTICK()` Returns a double precision value equal to the number of seconds between successive clock ticks.

OpenMP Environment Variables

TABLE E-7 and TABLE E-8 summarize the OpenMP Fortran API environment variables that control the execution of OpenMP programs.

TABLE E-7 Summary of OpenMP Fortran Environment Variables

`OMP_SCHEDULE` Sets schedule type for `DO` and `PARALLEL DO` directives specified with schedule type `RUNTIME`. If not defined, a default value of `STATIC` is used. Value is *"type*[*,chunk*]*"* Example: `setenv OMP_SCHEDULE "GUIDED,4"`.
`OMP_NUM_THREADS` Sets the number of threads to use during execution, unless set by a `NUM_THREADS` clause, or a call to `OMP_SET_NUM_THREADS`() subroutine. If not set, a default of 1 is used. Value is a positive integer. (*Current maximum is 128*). Example: `setenv OMP_NUM_THREADS 16`
`OMP_DYNAMIC` Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. If not set, a default value of `TRUE` is used. Value is `TRUE` or `FALSE`. Example: `setenv OMP_DYNAMIC FALSE`
`OMP_NESTED` Enables or disables nested parallelism. (*Nested parallelism is not supported.*) Value is `TRUE` or `FALSE`. The default, if not set, is `FALSE`. Example: `setenv OMP_NESTED TRUE`

TABLE E-8 Environment variables not part of the OpenMP Fortran API

`SUNW_MP_WARN` Controls warning messages issued by the runtime library. If set `TRUE`, the runtime library issues warning messages to `stderr`; `FALSE` disables warning messages. The default is `FALSE`. Example: `setenv SUNW_MP_WARN TRUE`
`SUNW_MP_THR_IDLE` Controls the end-of-task status of each thread executing the parallel part of a program. You can set the value to `spin`, `sleep` *n*`s`, or `sleep` *n*`ms`. The default is `SPIN` — a thread should spin (or busy-wait) after completing a parallel task, until a new parallel task arrives. Choosing `SLEEP` *time* specifies the amount of time a thread should spin-wait after completing a parallel task. If, while a thread is spinning, a new task arrives for the thread, the tread executes the new task immediately. Otherwise, the thread goes to sleep and is awakened when a new task arrives. *time* may be specified in seconds, (*n*`s`), or just (*n*), or milliseconds, (*n*`ms`). `SLEEP` with no argument puts the thread to sleep immediately after completing a parallel task. `SLEEP`, `SLEEP (0)`, `SLEEP (0s)`, and `SLEEP (0ms)` are all equivalent. Example: `setenv SUNW_MP_THR_IDLE (50ms)`
`STACKSIZE` Sets the thread stack size. The value is in kilobytes. Example: `setenv STACKSIZE 8192` sets the thread stack size to 8Mb.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W