Fortran User's Guide

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part No. 819-0492-10
January 2005, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Tables

Before You Begin

The *Fortran User's Guide* describes the environment and command-line options for the Sun™ Studio Fortran 95 compiler `f95`.

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

See also the companion *Fortran Programming Guide* for essential information on input/output, program development, libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and interoperability.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

TABLE P-1 Typeface Conventions *(Continued)*

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

TABLE P-2 Code Conventions

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

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

```
ΔΔ36.001
```

- The FORTRAN 77 standard used an older convention, spelling the name "FORTRAN" capitalized. The current convention is to use lower case: "Fortran 95"
- References to online man pages appear with the topic name and section number. For example, a reference to the library routine `GETENV` will appear as `getenv`(3F), implying that the man command to access this man page would be:
  `man -s 3F getenv`

Shell Prompts

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

Supported Platforms

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

In this document, the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Software and Man Pages

The compilers and tools and their man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the compilers and tools, you must have your `PATH` environment variable set correctly (see "Accessing the Compilers and Tools" on page xx). To access the man pages, you must have the your `MANPATH` environment variable set correctly (see "Accessing the Man Pages" on page xxi.).

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 variable and MANPATH variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun Studio compilers and tools are installed in the /opt directory. If your software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

Accessing the Compilers and Tools

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

▼ To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **Review the output to find a string of paths that contain /opt/SUNWspro/bin/.**

   If you find the path, your PATH variable is already set to access the compilers and tools. If you do not find the path, set your PATH environment variable by following the instructions in the next procedure.

▼ To Set Your PATH Environment Variable to Enable Access to the 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. If you have Forte Developer software, Sun ONE Studio software or another release of Sun Studio software installed, add the following path before the paths to those installations.**

   /opt/SUNWspro/bin

Accessing the Man Pages

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

▼ To Determine Whether You Need to Set Your MANPATH Environment Variable

1. **Request the dbx man page by typing the following at a command prompt.**

```
% man dbx
```

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

   If the dbx(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 procedure for setting your MANPATH environment variable.

▼ To Set Your MANPATH Environment Variable to Enable Access to the 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 the Integrated Development Environment

The Sun Studio integrated development environment (IDE) provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, or Fortran application.

The command to start the IDE is sunstudio. For details on this command, see the sunstudio(1) man page.

The correct operation of the IDE depends on the IDE being able to find the core platform. The sunstudio command looks for the core platform in two locations:

- The command looks first in the default installation directory, /opt/netbeans/3.5V.

- If the command does not find the core platform in the default directory, it assumes that the directory that contains the IDE and the directory that contains the core platform are both installed in or mounted to the same location. For example, if the path to the directory that contains the IDE is `/foo/SUNWspro`, the command looks for the core platform in `/foo/netbeans/3.5V`.

If the core platform is not installed or mounted to either of the locations where the `sunstudio` command looks for it, then each user on a client system must set the environment variable `SPRO_NETBEANS_HOME` to the location where the core platform is installed or mounted (*/installation_directory*`/netbeans/3.5V`).

Each user of the IDE also must add */installation_directory*`/SUNWspro/bin` to their `$PATH` in front of the path to any other release of Forte Developer software, Sun ONE Studio software, or Sun Studio software.

The path */installation_directory*`/netbeans/3.5V/bin` should not be added to the user's `$PATH`.

Accessing Compilers and Tools Documentation

You can access the documentation at the following locations:

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

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

- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed software only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialogs, in the IDE.

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 that is installed with the software on your local system or network.

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

Documentation in Accessible Formats

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

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Online help	HTML available through the Help menu in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Compilers and Tools Documentation

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

Document Title	Description
Fortran Programming Guide	Describes how to write effective Fortran code on Solaris environments; input/output, libraries, performance, debugging, and parallel processing.
Fortran Library Reference	Details the Fortran library and intrinsic routines
Fortran User's Guide	Describes the compile-time environment and command-line options for the `f95` compiler. Also includes guidelines for migrating legacy `f77` programs to `f95`.
C User's Guide	Describes the compile-time environment and command-line options for the `cc` compiler.
C++ User's Guide	Describes the compile-time environment and command-line options for the `CC` compiler.
OpenMP API User's Guide	Summary of the OpenMP multiprocessing API, with specifics about the implementation.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

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

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

Resources for Developers

Visit `http://developers.sun.com/prodtech/cc` to find these frequently updated resources:

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

You can find additional resources for developers at `http://developers.sun.com`.

Contacting Sun Technical Support

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

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

Sending Your Comments

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

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

Please include the part number (819-0492-10) of your document in the subject line of your email.

CHAPTER 1

Introduction

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

The `f95` compiler also provides a Fortran 77 compatibility mode that accepts most legacy Fortran 77 source codes. There is no longer a separate Fortran 77 compiler. See Chapter 5 for information on FORTRAN 77 compatibility and migration issues.

1.1 Standards Conformance

- `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 is based on IEEE standard 754-1985, and international standard IEC 60559:1989.
- `f95` provides support for the optimization-exploiting features of the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T, on Solaris platforms.
- In this document, “Standard” means conforming to the versions of the standards listed above. “Non-standard” or “Extension” refers to features that go beyond these versions of these standards.

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

1.2 Features of the Fortran 95 Compiler

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

- Global program checking across routines for consistency of arguments, commons, parameters, and the like.
- Optimized automatic and explicit loop parallelization for multiprocessor systems.
- VAX/VMS Fortran extensions, including:
  - Structures, records, unions, maps
  - Recursion
- OpenMP parallelization directives.
- Cray-style parallelization directives, including `TASKCOMMON`.
- 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 using `%VAL`.
- Compatibility between Fortran 77 and Fortran 95 programs and object binaries.
- Interval Arithmetic programming.
- Some "Fortran 2003" features, including Stream I/O.

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

1.3 Other Fortran Utilities

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

- **Sun Studio Performance Analyzer** — In depth performance analysis tool for single threaded and multi-threaded applications. See `analyzer`(1).
- **`asa`** — This Solaris utility is a Fortran output filter for printing files that have Fortran carriage-control characters in column one. Use `asa` to transform files formatted with Fortran carriage-control conventions into files formatted according to UNIX line-printer conventions. See `asa`(1).

- **`fdumpmod`** — A utility to display the names of modules contained in a file or archive. See `fdumpmod`(1).
- **`fpp`** — A Fortran source code preprocessor. See `fpp`(1).
- **`fsplit`** — This utility splits one Fortran file of several routines into several files, each with one routine per file. Use `fsplit` on FORTRAN 77 or Fortran 95 source files. See `fsplit`(1)

1.4 Debugging Utilities

The following debugging utilities are available:

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

1.5 Sun Performance Library

The Sun Performance Library™ is a library of optimized subroutines and functions for computational linear algebra and Fourier transforms. It is based on the standard libraries LAPACK, BLAS1, BLAS2, BLAS3, FFTPACK, VFFTPACK, and LINPACK generally available through Netlib (`www.netlib.org`).

Each subprogram in the Sun Performance Library performs the same operation and has the same interface as the standard library versions, but is generally much faster and accurate and can be used in a multiprocessing environment.

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

1.6 Interval Arithmetic

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

See the *Fortran 95 Interval Arithmetic Programming Reference* for details.

1.7 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 Studio 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:

`f95`(1)	The Fortran 95 command-line options
`analyzer`(1)	Performance Analyzer
`asa`(1)	Fortran carriage-control print output post-processor
`dbx`(1)	Command-line interactive debugger
`fpp`(1)	Fortran source code pre-processor
`cpp`(1)	C source code pre-processor

fdumpmod(1)	Display contents of a MODULE (.mod) file.
fsplit(1)	Pre-processor splits Fortran source routines into single files
ieee_flags(3M)	Examine, set, or clear floating-point exception bits
ieee_handler(3M)	Handle floating-point exceptions
matherr(3M)	Math library error handling routine
ild(1)	Incremental link editor for object files
ld(1)	Link editor for object files

1.8 README Files

The 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: /opt/SUNWspro/READMEs/.

TABLE 1-1 READMEs of Interest

README File	Describes...
fortran_95	new and changed features, known limitations, documentation errata for this release of the Fortran 95 compiler, f95.
fpp_readme	overview of fpp features and capabilities
interval_arithmetic	overview of the interval arithmetic features in f95
math_libraries	optimized and specialized math libraries available.
profiling_tools	using the performance profiling tools, prof, gprof, and tcov.
runtime_libraries	libraries and executables that can be redistributed under the terms of the End User License.
performance_library	overview of the Sun Performance Library

The README file for each compiler is easily viewed by the -xhelp=readme command-line option. For example, the command:

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

displays the fortran_95 README file directly.

1.9 Command-Line Help

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

```
%f95 -help=flags
Items within [ ] are optional. Items within < > are variable
parameters.
Bar | indicates choice of literal values.
-someoption[={yes|no}] implies -someoption is equivalent to
-someoption=yes
_____________________________________________________________________
-a                                Collect data for tcov basic
block profiling
-aligncommon[=<a>] Align common block elements to the specified
                   boundary requirement; <a>={1|2|4|8|16}
-ansi                             Report non-ANSI extensions.
-autopar           Enable automatic loop parallelization
-Bdynamic          Allow dynamic linking
-Bstatic           Require static linking
-C                 Enable runtime subscript range checking
-c                 Compile only; produce .o files but suppress
                   linking
...etc.
```

CHAPTER 2

Using Fortran 95

This chapter describes how to use the Fortran 95 compiler.

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 Fortran 95 compiler can also be used to:

- Generate a parallelized executable file for multiple processors (`-openmp`).
- 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`).
- Check source code for ANSI standards conformance (`-ansi`).

2.1 A Quick Start

This section provides a quick overview of how to use the Fortran 95 compiler to compile and run Fortran programs. A full reference to command-line options appears in the next chapter.

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% f95 greetings.f
demo% a.out
 Real programmers write Fortran!
demo%
```

In this example, `f95` 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% f95 -o greetings greetings.f
demo% greetings
 Real programmers write Fortran!
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 at a shell prompt.

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

2.2 Invoking the Compiler

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

`f95` [*options*] *files...*

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 Fortran 95 compiler with either the **`f95`** or **`f90`** command.

New: The compiler will also accept source files with the extension `.f03` or `.F03`. These are treated as equivalent to `.f95` and `.F95` and could be used as a way to indicate that a source file contains Fortran 2003 extensions.

Section 2.2.2, "Command-Line File Name Conventions" on page 2-4, describes the various source file extensions accepted by the compiler.

2.2.1 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.

2.2.2 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 Filename Suffixes Recognized by the Fortran 95 Compiler

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`
`.f03`	Fortran 2003 free-format	Same action as `.f`
`.for`	Fortran 77 or Fortran 95	Same action as `.f`.
`.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.
`.F03`	Fortran 2003 free-format	Same as `.F95`
`.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 Chapter 4.

2.2.3 Source Files

The Fortran compiler 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.

2.2.4 Source File Preprocessors

`f95` supports 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 compiler 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 on files with a `.F`, `.F90`, `F95`, or `.F03` 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.

2.2.5 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).

2.2.6 Consistent Compiling and Linking

Ensuring a consistent choice of compiling and linking options is critical whenever compilation and linking are done in separate steps. Compiling any part of a program with some options requires linking with the same options. Also, a number of options require that *all* source files be compiled with that option, *including* the link step.

The option descriptions in Chapter 3 identify such options.

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

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

2.2.7 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 `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!

2.2.8 Fortran 95 Modules

`f95` automatically creates module information 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` .

See the `-M`*path* and `-moddir` *dirlist* option flags for information on how to set the defaults paths for writing and searching for module information files.

See also the `-use` compiler option for implicitly invoking `MODULE` declarations in all compilation units.

Use the `fdumpmod`(1) command to display information about the contents of a `.mod` module information file.

For detailed information, see Section 4.9, “Module Files” on page 4-23.

2.3 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 compiler 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 Section 4.8, “Directives” on page 4-21. A complete summary of all the directives recognized by `f95` appears in Appendix D.

Note – Directives are not part of the Fortran standard.

2.3.1 General Directives

The various forms of a general Fortran 95 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.
- Directives using the `!` comment-indicator character may appear in any position on the line 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 `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 compiler recognize the following general directives:

TABLE 2-2 Summary of General Fortran Directives

`C` *Directive*	`C$PRAGMA C(`*list*`)` Declares a list of names of external functions as C language routines.
`IGNORE_TKR` *Directive*	`C$PRAGMA IGNORE_TKR` {*name* {, *name*} ...} The compiler ignores the type, kind, and rank of the specified dummy argument names appearing in a generic procedure interface when resolving a specific call.
`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.)

`ASSUME` *Directives*	`C$PRAGMA` [`BEGIN`] `ASSUME (`*expression* [`,` *probability*]`)`
	`C$PRAGMA END ASSUME`
	Make assertions about conditions at certain points in the program that the compiler can assume are true.

2.3.1.1 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)
```

2.3.1.2 The IGNORE_TKR Directive

This directive causes the compiler to ignore the type, kind, and rank of the specified dummy argument names appearing in a generic procedure interface when resolving a specific call.

For example, in the procedure interface below, the directive specifies that SRC can be any data type, but LEN can be either KIND=4 or KIND=8.

The interface block defines two specific procedures for a generic procedure name. This example is shown in Fortran 95 free format.

```
INTERFACE BLCKX

SUBROUTINE BLCK_32(LEN,SRC)
  REAL SRC(1)
!$PRAGMA IGNORE_TKR SRC
  INTEGER (KIND=4) LEN
END SUBROUTINE

SUBROUTINE BLCK_64(LEN,SRC)
  REAL SRC(1)
!$PRAGMA IGNORE_TKR SRC
  INTEGER (KIND=8) LEN
END SUBROUTINE

END INTERFACE
```

The subroutine call:

```
INTEGER L
REAL S(100)
CALL BLCKX(L,S)
```

The call to BLCKX will call BLCK_32 when compiled normally, and BLCK_64 when compiled with -xtypemap=integer:64. The actual type of S does not determine which routine to call. This greatly simplifies writing generic interfaces for wrappers that call specific library routines based on argument type, kind, or rank.

Note that dummy arguments for assumed-shape arrays, Fortran pointers, or allocatable arrays cannot be specified on the directive. If no names are specified, the directive applies to all dummy arguments to the procedure, except dummy arguments that are assumed-shape arrays, Fortran pointers, or allocatable arrays.

2.3.1.3 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
```

2.3.1.4 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.

2.3.1.5 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 f95 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.

2.3.1.6 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.

2.3.1.7 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.*

2.3.1.8 The PREFETCH Directives

The `-xprefetch` option flag, Section , "-xprefetch[=a[,a]]" on page 3-85, 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.

2.3.1.9 The ASSUME Directives

The `ASSUME` directive gives the compiler hints about conditions at certain points in the program. These assertions can help the compiler to guide its optimization strategies. The programmer can also use these directives to check the validity of the program during execution. There are two formats for `ASSUME`.

The syntax of the "point assertion" `ASSUME` is

```
C$PRAGMA ASSUME (expression [,probability])
```

Alternatively, the "range assertion" ASSUME is:

```
C$PRAGMA BEGIN ASSUME [expression [, probability)
      block of statements
C$PRAGMA END ASSUME
```

Use the point assertion form to state a condition that the compiler can assume at that point in the program. Use the range assertion form to state a condition that holds over the enclosed range of statements. The BEGIN and END pairs in a range assertion must be properly nested.

The required *expression* is a boolean expression that can be evaluated at that point in the program that does not involve user-defined operators or function calls except for those listed below.

The optional *probability* value is a real number from 0.0 to 1.0, or an integer 0 or 1, giving the probability of the expression being true. A probability of 0.0 (or 0) means never true, and 1.0 (or 1) means always true. If not specified, the expression is considered to be true with a high probability, but not a certainty. An assertion with a probability other than exactly 0 or 1 is a *non-certain assertion*. Similarly, an assertion with a probability expressed exactly as 0 or 1 is a *certain assertion*.

For example, if the programmer knows that the length of a DO loop is always greater than 10,000, giving this hint to the compiler can enable it to produce better code. The following loop will generally run faster with the ASSUME pragma than without it.

```
C$PRAGMA BEGIN ASSUME(__tripcount().GE.10000,1) !! a big loop
        do i = j, n
           a(i) = a(j) + 1
        end do
C$PRAGMA END ASSUME
```

Two intrinsic functions are available for use specifically in the expression clause of the ASSUME directive. (Note that their names are prefixed by two underscores.)

`__branchexp()`	Use in point assertions placed immediately before a branching statement with a boolean controlling expression. It yields the same result as the boolean expression controlling the branching statement.
`__tripcount()`	Yields the trip count of the loop immediately following or enclosed by the directive. When used in a point assertion, the statement following the directive must be the first line of a DO. When used in a range assertion, it applies to the outermost enclosed loop.

This list of special intrinsics might expand in future releases.

Use with the -xassume_control compiler option. (See Section , "-xassume_control[=keywords]" on page 3-61) For example, when compiled with -xassume_control=check, the example above would produce a warning if the trip count ever became less than 10,000.

Compiling with -xassume_control=retrospective will generate a summary report at program termination of the truth or falsity of all assertions. See the f95 man page for details on -xassume_control.

Another example:

```
C$PRAGMA ASSUME(__tripcount.GT.0,1)
       do i=n0, nx
```

Compiling the above example with -xassume_control=check will issue a runtime warning should the loop not be taken because the trip count is zero or negative.

2.3.2 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 -openmp, -parallel, or -explicitpar are used. Details regarding Fortran parallelization can be found in the *OpenMP API User's Guide* and the *Fortran Programming Guide*.

The Fortran compiler supports the OpenMP shared memory parallelization model, as well as legacy Sun and Cray directives.

Parallelization features of the compiler are not available currently on x86 platforms.

2.3.2.1 OpenMP Parallelization Directives

The Fortran 95 compiler recognizes the OpenMP Fortran shared memory multiprocessing API as the preferred parallel programming model. The API is specified by the OpenMP Architecture Review Board (http://www.openmp.org).

You must compile with the command-line option -openmp, to enable OpenMP directives. (See Section , "-openmp[={parallel|noopt|none}]" on page 3-41.)

For more information about the OpenMP directives accepted by f95, see the *OpenMP API User's Guide*.

2.3.2.2 Legacy Sun/Cray Parallelization Directives

Sun style parallelization directives are the default for `-parallel` and `-explicitpar`. 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. See also the *OpenMP API User's Guide* for guidelines on converting legacy Sun/Cray parallelization directives to OpenMP directives.

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
```

2.4 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.)

2.5 Compiler Usage Tips

The next sections suggest a number of ways to use the Fortran 95 compiler efficiently. A complete compiler options reference follows in the next chapter.

2.5.1 Determining Hardware Platform (SPARC)

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

```
demo% fpversion
A SPARC-based CPU is available.
 Kernel says CPU's clock rate is 750.0 MHz.
 Kernel says main memory's clock rate is 150.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.

 Use "-xtarget=ultra3" 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. `fpversion` is not available on x86 platforms.

2.5.2 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:

`f95 -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.

2.5.3 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 processor running the compiler should have at least 64 megabytes of memory; 256 megabytes are recommended. Enough swap space should also be allocated. 200 megabytes is the minimum; 300 megabytes is 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).

2.5.3.1 Swap Space Limits

The 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
```

2.5.3.2 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
```

2.5.3.3 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 `f95` compiler.

- A description of the syntax used for compiler option flags starts at Section 3.1, "Command Syntax" on page 3-1.
- Summaries of options arranged by functionality starts at Section 3.3, "Options Summary" on page 3-3.
- The complete reference detailing each compiler option flag starts at Section 3.4, "Options Reference" on page 3-11.

3.1 Command Syntax

The general syntax of the compiler command line is:

```
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`).

3.2 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 (braces) contain a set of choices for a required option.	`-d{y|n}` `-dy`
|	The "pipe" or "bar" symbol separates arguments, only *one* of which may be chosen.	`-B{dynamic|static}` `-Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`-R`*dir*`[:`*dir*`]` `-R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`-xinline=`*f1*`[,…`*fn*`]` `-xinline=alpha,dos`

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). This 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.
- In an optional list of choices, such as `-xhasc[={yes|no}]`, the first choice listed is the value assumed when the option flag appears on the command line without a value. For example, `-xhasc` is equivalent to `-xhasc=yes`.
- Source files, object files, and libraries are compiled and linked in the order in which they appear on the command line.

3.3 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.

Note that not all options are available on both SPARC and x86 platforms. Check the detailed reference section for availability.

The following table summarizes the `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`
Support Fortran 77 extensions and compatibility	`-f77`
Specify path for writing compiled `.mod` Module files	`-moddir=`*path*
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=`*path*

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

Function	Option Flag
Show elapsed time for each compilation phase	`-time`
Show version number of compiler and its phases	`-V`
Verbose messages	`-v`
Specify non-standard aliasing situations	`-xalias=`*list*
Compile with multiple processors	`-xjobs=`*n*
Compiled Code:	
Add/suppress trailing underscores on external names	`-ext_names=`*x*
Inline specified user functions	`-inline=`*list*
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 with `dbx`	`-g`
Compile for browsing with source browser	`-sb, -sbfast`
Flag use of undeclared variables	`-u`
Check `C$PRAGMA ASSUME` assertions	`-xassume_control=check`
Check for stack overflow at runtime	`-xcheck=stkovf`
Enable runtime task common check	`-xcommonchk`

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

Function	Option Flag
Compile for Performance Analyzer	`-xF`
Generate cross-reference 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`
Show/suppress warning messages	`-w`*n*
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`*path*
Link with library `lib`*name*`.a` or `lib`*name*`.so`	`-l`*name*
Build runtime library search path into executable	`-R`*path*
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`

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

Function	Option Flag
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*
Specify rounding method for formatted input/output	`-iorounding=`*mode*
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*`]`
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*`]`
Enable/adjust compiler generated prefetch instructions	`-xprefetch=`*list*
Control automatic generation of prefetch instructions	`-xprefetch_level=`*n*
Enable generation or use of performance profiling data	`-xprofile=`*p*
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*`]`

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

Function	Option Flag
Parallelization:	
Enable automatic parallelization of DO loops	`-autopar`
Enable parallelization of loops explicitly marked with directives	`-explicitpar`
Show loop parallelization information	`-loopinfo`
Specify Cray-style parallelization directives	`-mp=CRAY`
Compile for hand-coded multithreaded programming	`-mt`
Accept OpenMP API directives and set appropriate environment	`-openmp[=`*keyword*`]`
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*
Accept extended (132 character) source lines	`-e`
Apply preprocessor to `.F` and/or `.F90` and `.F95` files but do not compile	`-F`
Accept Fortran 95 fixed-format input	`-fixed`
Preprocess all source files with the `fpp` preprocessor	`-fpp`
Accept Fortran 95 free-format input	`-free`
Add directory to include file search path	`-I`*path*
Add directory to module search path	`-M`*path*
Recognize upper and lower case as distinct	`-U`
Tread hollerith as character in actual arguments	`-xhasc={yes|no}`
Select preprocessor, `cpp` or `fpp`, to use	`-xpp[={fpp|cpp}]`
Allow recursive subprogram calls	`-xrecursive`
Target Platform:	
Specify target platform instruction set for the optimizer	`-xarch=`*a*

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

Function	Option Flag
Specify target cache properties for optimizer	`-xcache=`*a*
Specify target processor for the optimizer	`-xchip=`*a*
Specify target platform for the optimizer	`-xtarget=`*a*

3.3.1 Commonly Used Options

The compiler has 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 debugging.	`-g`
Performance—invoke the optimizer to produce faster running programs.	`-O[`*n*`]`
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 source code.	`-fixed`

3.3.2 Macro Flags

Some option flags are macros that expand into a specific set of other flags. These are provided as a convenient way to specify a number of options that are usually expressed together to select a certain feature.

TABLE 3-5 Macro Option Flags

Option Flag	Expansion
`-dalign`	`-xmemalign=8s -aligncommon=16`
`-f`	`-aligncommon=16`
`-fast`	`-xO5 -libmil -fsimple=2 -dalign -xlibmopt -depend -fns -ftrap=common -pad=local -xvector=yes -xprefetch=yes` *(SPARC)* `-xprefetch_level=2` *(SPARC)* `-nofstore` *(x86)*
`-fnonstd`	`-fns -ftrap=common`
`-parallel`	`-autopar -explicitpar -depend`
`-xia=widestneed`	`-xinterval=widestneed -ftrap=%none -fns=no -fsimple=0`
`-xia=strict`	`-xinterval=strict -ftrap=%none -fns=no -fsimple=0`
`-xtarget`	`-xarch=`*a* `-xcache=`*b* `-xchip=`*c*

Settings that follow the macro flag on the command line override the expansion of the macro. For example, to use `-fast` but with an optimization level of `-O3`, the `-O3` must come after `-fast` on the command line.

3.3.3 Backward Compatibility and Legacy Options

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

TABLE 3-6 Backward Compatibility Options

Action	Option
Allow assignment to constant arguments.	`-copyargs`
Treat hollerith constant as character or typeless in call argument lists.	`-xhasc[={yes|no}]`
Support Fortran 77 extensions and conventions	`-f77`
Nonstandard arithmetic—allow nonstandard arithmetic.	`-fnonstd`

TABLE 3-6 Backward Compatibility Options *(Continued)*

Action	Option
Optimize performance for the host system.	`-native`
`DO` loops—use one trip `DO` loops.	`-onetrip`
Allow legacy aliasing situations	`-xalias=`*keywords*

Use of these option flags is not recommended for producing portable Fortran 95 programs.

3.3.4 Obsolete Option Flags

The following options are considered obsolete and should not be used. They might be removed from later releases of the compiler.

TABLE 3-7 Obsolete `f95` Options

Option Flag	Equivalent
`-a`	`-xprofile=tcov`
`-cg89`	`-xtarget=ss2`
`-cg92`	`-xtarget=ss1000`
`-native`	`-xtarget=native`
`-noqueue`	License queueing. No longer needed.
`-p`	Profiling. Use `-pg` or the Performance Analyzer
`-pic`	`-xcode=pic13`
`-PIC`	`-xcode=pic32`
`-sb`	No longer needed.
`-sbfast`	No longer needed.
`-silent`	No longer needed.

3.4 Options Reference

This section describes all of the `f95` compiler command–line option flags, including various risks, restrictions, caveats, interactions, examples, and other details.

Unless indicated otherwise, each option is valid on both SPARC and x86 platforms. Option flags valid only on SPARC platforms are marked **(SPARC)**. Option flags valid only on x86 platforms are marked **(x86)**.

Option flags marked ***(Obsolete)*** are obsolete and should not be used. In many cases they have been superceded by other options or flags that should be used instead.

`-a`

(Obsolete) Profile by basic block using `tcov`, old style.

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, *Program Performance Analysis Tools*.

`-aligncommon`[`=`{`1`|`2`|`4`|`8`|`16`}]

Specify the alignment of data in common blocks and standard numeric sequence types.

The value indicates the maximum alignment (in bytes) for data elements within common blocks and standard numeric sequence types.

Note – A *standard numeric sequence type* is a derived type containing a `SEQUENCE` statement and only default component data types (`INTEGER, REAL, DOUBLEPRECISION, COMPLEX` without `KIND=` or `*`*size*). Any other type, such as `REAL*8`, will make the type non-standard.

For example, `-aligncommon=4` would align 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 compiler aligns elements in common blocks and numeric sequence types on (at most) 4-byte boundaries.

Specifying `-aligncommon` without a value defaults to 1 - all common block and numeric sequence type elements align 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.

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

-arg=local

Preserve actual arguments over `ENTRY` statements.

When you compile a subprogram with alternate entry points with this option, `f95` uses copy/restore to preserve the association of dummy and actual arguments.

This option is provided for compatibility with legacy Fortran 77 programs. Code that relies on this option is non-standard.

-autopar

Enable automatic loop parallelization.

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`.

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 in Section , "-mt" on page 3-37.

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.

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:

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

These are loader and linker options. Compiling and linking in separate steps with `-Bx` 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 and conformance at runtime.

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. `-C` also adds runtime checks for array conformance in array syntax expressions

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.

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

(*Obsolete*, SPARC) Compile for generic SPARC architecture.

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

-cg92

(*Obsolete*, SPARC) Compile for SPARC V8 architecture.

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.

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.

This option only applies to `.F`, `.F90`, `.F95`, and `.F03` 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. (Note the two leading underscores.)

- The product version is predefined (in hex) in `__SUNPRO_F90`, and `__SUNPRO_F95`.
  For example `__SUNPRO_F95` is `0x810` for the Sun Studio 10 release.
- The following macros are predefined on appropriate systems:

  `__sparc`, `__unix`, `__sun`, `__SVR4`, `__i386` ,
  `__SunOS_5_6`, `__SunOS_5_7`, `__SunOS_5_8`, `__SunOS_5_9`, `__SunOS_5_10`

  For instance, the value `__sparc` is defined on SPARC systems.
- The following are predefined with no underscores, but they might be deleted in a future release: `sparc`, `unix`, `sun`
- On SPARC V9 systems, the `__sparcv9` macro is also defined.
- On 64-bit x86 systems, the macros `__amd64` and `__x86_64` are defined.

Compile with the verbose option (`-v`) to see the definitions created by the compiler.

You can use these values in such preprocessor conditionals as the following:

```
#ifdef __sparc
```

`f95` uses 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 blocks and standard numerical sequence types, and generate faster multi-word load/stores.

This flag changes the data layout in COMMON blocks, numeric sequence types, 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` *on SPARC platforms*

`-aligncommon=8` *on 32-bit x86 platforms*

`-aligncommon=16` *on 64-bit x86 platforms*.

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

Note that because `-dalign` invokes `-aligncommon`, standard numeric sequence types are also affected by this option. See Section , "-aligncommon[={1|2|4|8|16}]" on page 3-11

-dbl_align_all[={yes|no}]

Force alignment of data on 8–byte boundaries

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.

Use with -dalign to enable added efficiency with multi-word load/stores.

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

-depend[={yes|no}]

Analyze loops for data dependencies and do loop restructuring.

Dependence analysis is enabled with -depend or -depend=yes. The analysis is disabled with -depend=no, which is the compiler default.

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. Note also that specifying an optimization level -O3 or higher automatically adds -depend. (See the *Fortran Programming Guide*.)

-dn

Disallow dynamic libraries. See Section , "-d{y|n}" on page 3-17.

-dryrun

Show commands built by the f95 command-line driver, but do not compile.

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.

- -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 and 32-bit x86 Solaris platforms, and all 32-bit Solaris platforms starting with the Solaris 10 release. Applications must link with the dynamic libraries in these cases.

-e

Accept extended length input source line.

Extended source lines can be 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[={%all | %none | *taglist*}]

Suppress warning messages listed by tag name.

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

Example:

```
f95 -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.

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% f95 -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`.

-errwarn[={%all | %none | *taglist*}]

Treat warning messages as errors.

The taglist specifies a list of comma-separated tag names of warning messages that should be treated as errors. If `%all`, treat all warnings as errors. If `%none`, no warnings are treated as errors.

See also `-errtags`.

-explicitpar

Parallelize loops explicitly marked by Sun or Cray directives.

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 and/or Cray explicit parallelization directives. DO loops immediately preceded by parallelization directives will have threaded code generated for them.

To enable OpenMP explicit parallelization directives, do not use `-explicitpar`. Use `-openmp` instead. See Section , "-openmp[={parallel|noopt|none}]" on page 3-41)

Note – `-explicitpar` 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 (Section , "-mp={%none|sun|cray}" on page 3-36) to select the style of parallelization directives enabled. The default with -explicitpar is Sun directives. Use -explicitpar -mp=cray to enable Cray directives.

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.

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. Use this flag to allow Fortran 95 routines to call (and be called by) other programming language routines.

-F

Invoke the source file preprocessor, but do not compile.

Apply the fpp preprocessor to .F, .F90, .F95, and .F03 source files listed on the command line, and write the processed result on a file with the same name but with filename extension changed to .f (or .f95 or .f03), but do not compile.

Example:

```
f95 -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 double- and quad-precision data in COMMON blocks.

-f is a legacy option flag equivalent to -aligncommon=16. Use of -aligncommon is preferred.

The default alignment of data in COMMON blocks is on 4-byte boundaries. -f changes the data layout of double- and quad-precision data in COMMON blocks and EQUIVALENCE classes to be placed 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 SPARC environments with `-xarch=v9` or `v9a`).

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.

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 Section , "-dalign" on page 3-16. Because `-dalign` is part of the `-fast` option, so is `-f`.

`-f77`[`=`*list*]

Select Fortran 77 compatibility mode.

This option flag enables porting legacy Fortran 77 source programs, including those with language extensions accepted by the `f77` compiler, to the `f95` Fortran 95 compiler.

list is a comma-separated list selected from the following possible keywords:

keyword	meaning
`%all`	Enable all the Fortran 77 compatibility features.
`%none`	Disable all the Fortran 77 compatibility features.
`backslash`	Accept backslash as an escape sequence in character strings.
`input`	Allow input formats accepted by `f77`.
`intrinsics`	Limit recognition of intrinsics to only Fortran 77 intrinsics.
`logical`	Accept Fortran 77 usage of logical variables, such as: - assigning integer values to logical variables - allowing arithmetic expressions in logical conditional statements, with `.NE.0` representing `.TRUE.` - allowing relational operators `.EQ.` and `.NE.` with logical operands
`misc`	Allow miscellaneous `f77` Fortran 77 extensions.

keyword	meaning
`output`	Generate `f77`-style formatted output, including list-directed and `NAMELIST` output.
`subscript`	Allow non-integer expressions as array subscripts.
`tab`	Enable `f77`-style TAB-formatting, including unlimited source line length. No blank padding will be added to source lines shorter than 72 characters.

All keywords can be prefixed by `no%` to disable the feature, as in:

```
-f77=%all,no%backslash
```

The default, when `-f77` is not specified, is `-f77=%none`. Using `-f77` without a list is equivalent to specifying `-f77=%all`.

Exceptions Trapping and `-f77`:

Specifying `-f77` does not change the Fortran 95 trapping mode, which is `-ftrap=common`. Fortran 95 differs from the Fortran 77 compiler's behavior regarding arithmetic exception trapping. The Fortran 77 compiler allowed execution to continue after an arithmetic exception occurred. Compiling with `-f77` also causes the program to call `ieee_retrospective` on program exit to report on any arithmetic exceptions that might have occurred. Specify `-ftrap=none` following the `-f77` option flag on the command line to mimic the original Fortran 77 behavior.

See Chapter 5 for complete information on `f77` compatibility and Fortran 77 to Fortran 95 migration.

See also the `-xalias` flag for handling non-standard programming syndromes that may cause incorrect results.

-fast

Select options that optimize execution performance.

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` for any release.

`-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 **(SPARC)**
- -fns
- -fsimple=2
- -ftrap=common
- -libmil
- -xtarget=native
- -O5
- -xlibmopt
- -pad=local **(SPARC)**
- -xvector=yes **(SPARC)**
- -xprefetch=yes
- -xprefetch_level=2
- -nofstore **(x86)**

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:
  ```
  f95 -fast -xtarget=ultra ...
  ```
- The -O5 optimization level option.
- The -depend option analyzes loops for data dependencies and possible restructuring. **(SPARC)**
- 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 Section , "-fsimple[={1|2|0}]" on page 3-27.

- 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. **(SPARC)**
- `-xvector=yes` transforms certain math library calls within DO loops to single calls to a vectorized library equivalent routine with vector arguments.
- `-fns` selects non-standard floating-point arithmetic exception handling and gradual underflow. See Section , "`-fns[={yes|no}]`" on page 3-25.
- Trapping on common floating-point exceptions, `-ftrap=common`, is the enabled with Fortran 95.
- `-xprefetch=yes` enables the compiler to generate hardware prefetch instructions where appropriate.
- `-xprefetch_level=2` sets the default level for insertion of prefetch instructions.
- `-nofstore` cancels forcing expressions to have the precision of the result. **(x86)**

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.

Note that the set of options selected by the `-fast` flag can change with each compiler release.

-fixed

Specify fixed–format Fortran 95 source input files.

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

-flags

Synonym for `-help`.

-fnonstd

Initialize floating–point hardware to non–standard preferences.

This option is a macro 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[={yes|no}]

Select nonstandard floating–point mode.

The default is the 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` without a value is the same as `-fns=yes`.

This option flag enables nonstandard floating-point mode when the program begins execution. On SPARC platforms, 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.

On x86 platforms, this option is enabled only for Pentium III and Pentium 4 processors (sse or sse2 instruction sets).

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

-fpover[={**yes**|**no**}]

Detect floating-point overflow in formatted input.

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` without a value is equivalent to `-fpover=yes`.

-fpp

Force preprocessing of input with `fpp`.

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 Section , "-xpp={fpp|cpp}" on page 3-84.

-fprecision={single|double|extended}

(x86) Initialize non-default floating-point rounding precision mode.

On x86, sets the floating-point precision mode to either `single`, `double`, or `extended`.

With a value of `single` or `double`, this flag causes the rounding precision mode to be set to single or double precision respectively at program initiation. With `extended`, or by default when the `-fprecision` flag is not specified, the rounding precision mode is initialized to extended precision.

This option is effective only on x86 systems and only if used when compiling the main program.

-free

Specify free–format source input files.

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={nearest|tozero|negative|positive}

Set the IEEE rounding mode in effect at startup.

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 the value 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[={1|2|0}]

Select floating–point optimization preferences.

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.

The defaults are:

- Without the `-fsimple` flag, the compiler defaults to `-fsimple=0`
- With `-fsimple` without a value, the compiler uses `-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` selectes `-fsimple=2`.

-fstore

(x86) Force precision of floating-point expressions.

For assignment statements, this option forces all floating-point expressions to the precision of the destination variable. This is the default. However, the `-fast` option includes `-nofstore` to disable this option. Follow `-fast` with `-fstore` to turn this option back on.

-ftrap=*t*

Set floating–point trapping mode in effect at startup.

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`.

The `f95` default is `-ftrap=common`. This differs from the C and C++ compiler defaults, which is `-ftrap=none`.

Sets the IEEE 745 trapping mode in effect at startup but does not install a SIGFPE handler. You can use `ieee_handler`(3M) or `fex_set_handling`(3M) to simultaneously enable traps and install a SIGFPE handler. If you specify more than one value, the list is processed sequentially from left to right. The common exceptions, by definition, are invalid, division by zero, and 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` 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.

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.

Produce additional symbol table information for debugging with `dbx`(1) debugging utility and for performance analysis with the 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.

Note – In previous releases of the compiler, the `-g` option would make `-xildon` the default incremental linker option when only `.o` object files appear on the command line (see Section , "-xild{off|on}" on page 3-73). Starting with this release, you must compile with the `-xildon` flag to get this behavior.

To use the full capabilities of the 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 the manual *Sun Studio Performance Analyzer*.)

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.

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.

See also Section , "-xhelp={readme|flags}" on page 3-72.

-I*path*

Add *path* to the INCLUDE file search path.

Insert the directory path *path* at the start of the INCLUDE file search path. No space is allowed between -I and *path*. 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*path* 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 paths on INCLUDE or #include is:

1. The directory that contains the source file
2. The directories that are named in the -I options
3. The directories in the compiler's internal default list

4. /usr/include/

To invoke the preprocessor, you must be compiling source files with a .F, .F90, .F95, or .F03 suffix.

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

Enable or disable inlining of specified routines.

Request the optimizer to inline the user–written routines appearing in a comma-separated list of function and subroutine names. 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.

Specify %auto to enable automatic inlining at optimization levels -O4 or -O5. Automatic inlining at these optimization levels is normally turned off when explicit inlining is specified with -inline.

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 -xipo or -xcrossfile are 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.

-iorounding[={compatible|processor-defined}]

Set floating-point rounding mode for formatted input/output.

Sets the `ROUND=` specifier globally for all formatted input/output operations.

With `-iorounding=compatible`, the value resulting from data conversion is the one closer to the two nearest representations, or the value away from zero if the value is halfway between them.

With `-iorounding=processor-defined`, the rounding mode is the processor's default mode. This is the default when `-iorounding` is not specified.

`-Kpic`

(Obsolete) Synonym for `-xcode=pic13`.

`-KPIC`

(Obsolete) Synonym for `-xcode=pic32`.

`-L`*path*

Add *path* to list of directory paths to search for libraries.

Adds *path* to the *front* of the list of object–library search directories. A space between `-L` and *path* is optional. This option is passed to the linker. See also Section , "-lx" on page 3-33.

While building the executable file, `ld`(1) searches *path* for archive libraries (`.a` files) and shared libraries (`.so` files). `ld` searches *path* 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`*path*, see `ld`(1).

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

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

```
demo% f95 -L./dir1 -L./dir2 any.f
```

`-l`*x*

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

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 `libVZY`:

```
demo% f95 any.f -lVZY
```

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

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

```
demo% f95 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.

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 loop parallelization results.

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
```

-M*path*

Specify MODULE directory, archive, or file.

Look in path for Fortran 95 modules referenced in the current compilation. This path is searched in addition to the current directory.

path can specify a directory, .a archive file of precompiled module files, or a .mod precompiled module file. The compiler determines the type of the file by examining its contents.

An archive .a file must be explicitly specified on a -M option flag to be searched for modules. The compiler will not search archive files by default.

Only .mod files with the same names as the MODULE names appearing on USE statements will be searched. For example, the statement USE ME causes the compiler to look only for the module file me.mod

When searching for modules, the compiler gives higher priority to the directory where the module files are being written. This is controlled by the -moddir compiler option, or the MODDIR environment variable. When neither are specified, the default write-directory is the current directory. When both are specified, the write-directory is the path specified by the -moddir flag.

This means that if only the -M flag appears, the current directory will be searched for modules first before any object listed on the -M flag. To emulate the behavior of previous releases, use:

```
-moddir=empty-dir -Mdir -M
```

where *empty-dir* is the path to an empty directory.

There should be no space between the `-M` and the path. For example, `-M/home/siri/PK15/Modules`

See Section 4.9, "Module Files" on page 4-23 for more information about modules in Fortran 95.

-moddir=*path*

Specify where the compiler will write compiled `.mod` MODULE files.

The compiler will write the `.mod` MODULE information files it compiles in the directory specified by *path*. The directory path can also be specified with the `MODDIR` environment variable. If both are specified, this option flag takes precedence.

The compiler uses the current directory as the default for writing `.mod` files.

See Section 4.9, "Module Files" on page 4-23 for more information about modules in Fortran 95.

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

Select Sun or Cray parallelization directives.

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

The default with `-explicitpar` is `-mp=sun`.

`-mp=sun`	Accept Sun–style directives: `C$PAR` or `!$PAR` prefix.
`-mp=cray`	Accept Cray–style directives: `CMIC$` or `!MIC$` prefix.
`-mp=%none`	Ignore all parallelization directives.

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

```
-explicitpar -stackvar -mp=cray
```

To compile for OpenMP parallelization, use the `-openmp` flag. See Section , "`-openmp[={parallel|noopt|none}]`" on page 3-41.

Sun and Cray directives cannot both be active in the same compilation unit.

A summary of the Sun and Cray parallelization directives appears in Appendix D in this manual. See the *Fortran Programming Guide* for details.

-mt

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

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

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

-noautopar

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

-nodepend

(SPARC) Cancel any `–depend` appearing earlier on the command line.

-noexplicitpar

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

-nofstore

(x86) Cancel `-fstore` on command line.

The compiler default is `-fstore`. `-fast` includes `-nofstore`.

-nolib

Disable linking with system libraries.

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.

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.

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

```
demo% f95 -fast -nolibmil …
```

-noreduction

Disable `-reduction` on command line.

This option disables `-reduction`.

-norunpath

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

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.

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. Adds `-depend` automatically.

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.

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.

-onetrip

Enable one trip `DO` loops.

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[={parallel|noopt|none}]

Enable explicit parallelization with Fortran 95 OpenMP Version 2.0 directives.

The flag accepts the following optional keyword suboptions:

parallel	• Enables recognition of OpenMP pragmas, and the program is parallelized accordingly. • The minimum optimization level for -xopenmp=parallel is -xO3. The compiler changes the optimization from a lower level to -xO3 if necessary, and issues a warning. • Defines preprocessor token _OPENMP to be 200011 • Invokes -stackvar automatically.
noopt	• Enables recognition of OpenMP pragmas, and the program is parallelized accordingly. • The compiler does not raise the optimization level if it is lower than -xO3. If you explicitly set the optimization to a level lower than -xO3, as in -xO2 -openmp=noopt the compiler will issue an error. If you do not specify an optimization level with -openmp=noopt, the OpenMP pragmas are recognized, the program is parallelized accordingly, but no optimization is done. • Defines preprocessor token _OPENMP to be 200011 • Invokes -stackvar automatically.
none	Disables recognition of OpenMP pragmas and does not change the optimization level. (This is the compiler's default.)

-openmp specified without a suboption keyword is equivalent to -openmp=parallel. *Note that this default might change in later releases.*

To debug OpenMP programs with dbx, compile with -g -openmp=noopt to be able to breakpoint within parallel regions and display the contents of variables.

The OpenMP directives are summarized in the *OpenMP API User's Guide*.

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.

When compiling and linking in separate steps, also specify -openmp on the link step. This is especially important when compiling libraries that contain OpenMP directives.

-PIC

(*Obsolete*, SPARC) Compile position–independent code with 32-bit addresses.

`-PIC` is equivalent to `-xcode=pic32`. See Section , "-xcode=keyword" on page 3-65 for more information about position-independent code.

-p

(*Obsolete*) Compile for profiling with the `prof` profiler.

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.

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 `%none` or either (or both) `local` or `common`:

`local`	Add padding between adjacent *local* variables.
`common`	Add padding between variables in common blocks.
`%none`	Do not add padding. (Compiler default.)

If both `local` and `common` are specified, they can appear in any order.

Defaults for `-pad`:

- The compiler does no padding by default.
- Specifying `-pad`, but without a value is equivalent to `-pad=local,common`.

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 Section , "-stackvar" on page 3-47.

The program must conform to the following restrictions:

- 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.

It is the programmer's responsibility to make sure that common blocks are compiled consistently when -pad is used. Common blocks appearing in different program units that are compiled inconsistently with -pad=common will cause errors. Compiling with -Xlist will report when common blocks with the same name have different lengths in different program units.

-parallel

Parallelize with: -autopar, -explicitpar, -depend

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. See also Section , "-explicitpar" on page 3-19.

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

Sun-style parallelization directives are enabled by default. Use -mp=cray to select Cray style parallelization directives. (Note: For OpenMP parallelization use -openmp, not -parallel.)

Avoid -parallel if you do your own thread management. See Section , "-mt" on page 3-37.

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.

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 source 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`

(*Obsolete*, SPARC) Compile position–independent code for shared library.

`-pic` is equivalent to `-xcode=pic13`. See Section , "`-xcode=keyword`" on page 3-65 for more information on position-indepented code.

`-Qoption` *pr ls*

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`.

`-R` *ls*

Build dynamic library search paths into the executable file.

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, `/opt/SUNWspro/lib`, that is always searched last.

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

`-r8const`

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

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, see Section , "-xtypemap=spec" on page 3-95.

Use this option flag carefully. It could cause interface problems when a subroutine or function expecting a `REAL*4` argument is called with a `REAL*4` constant that gets promoted to `REAL*8`. It could also cause problems with programs reading unformatted data files written by an unformatted write with `REAL*4` constants on the I/O list.

-reduction

Recognize reduction operations in loops.

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% f95 -parallel -reduction any.f
```

-S

Compile and only generate assembly code.

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.

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

(*Obsolete*) Produce table information for the source code browser.

Note – `-sb` cannot be used on source files the compiler automatically passes through the `fpp` or `cpp` preprocessors (that is, files with `.F`, `.F90`, `.F95`, or `.F03` extensions), or used with the `-F` option.

-sbfast

(*Obsolete*) Produce *only* source code browser tables.

Produce *only* table information for the source code browser. 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`, `.F95`, or `.F03` extensions), or used with the `-F` option.

-silent

(*Obsolete*) Suppress compiler messages.

Normally, the `f95` compiler does not issue messages, other than error diagnostics, during compilation. This option flag is provided for compatibility with the legacy `f77` compiler, and its use is redundant except with the `-f77` compatibility flag.

-stackvar

Allocate local variables on the stack whenever possible.

This option makes writing recursive and re-entrant code easier and provides the optimizer more freedom when parallelizing loops.

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

Local variables are variables that are not dummy arguments, `COMMON` variables, variables inherited from an outer scope, or module variables made accessible by a `USE` statement.

With `-stackvar` in effect, local variables are allocated on the stack unless they have the attributes `SAVE` or `STATIC`. Note that explicitly initialized variables are implicitly declared with the `SAVE` attribute. A structure variable that is not explicitly initialized but some of whose components are initialized is, by default, not implicitly declared `SAVE`. Also, variables equivalenced with variables that have the `SAVE` or `STATIC` attribute are implicitly `SAVE` or `STATIC`.

A statically allocated variable is implicitly initialized to zero unless the program explicitly specifies an initial value for it. Variables allocated on the stack are not implicitly initialized except that components of structure variables can be initialized by default.

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 4 Megabytes (8 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.

Compile with `-xcheck=stkovf` to enable runtime checking for stack overflow situations. See Section , "-xcheck=keyword" on page 3-63.

-stop_status[={yes|no}]

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

The default is `-stop_status=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.

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.

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

-U

Recognize upper and lower case in source files.

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`. See the *Fortran Programming Guide* chapter on porting programs to Fortran 95.

-U*name*

Undefine preprocessor macro *name*.

This option applies only to source files that invoke the `fpp` or `cpp` pre-processor. 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.

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.

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 Section 2.3.1.3, "The `UNROLL` Directive" on page 2-11.

-use=*list*

Specify implicit `USE` modules.

list is a comma-separated list of module names or module file names.

Compiling with `-use=`*module_name* has the effect of adding a `USE` *module_name* statement to each subprogram or module being compiled. Compiling with `-use=`*module_file_name* has the effect of adding a `USE` *module_name* for each of the modules contained in the specified file.

See Section 4.9, "Module Files" on page 4-23 for more information about modules in Fortran 95.

-V

Show name and version of each compiler pass.

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.

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=*keywords*

Specify choice of VAX VMS Fortran extensions enabled.

The *keywords* specifier must be one of the following suboptions or a comma-delimited list of a selection of these.

`blank_zero`	**Interpret blanks in formatted input as zeros on internal files.**
`debug`	Interpret lines starting with the character 'D' to be regular Fortran statements rather than comments, as in VMS Fortran.
`rsize`	Interpret unformatted record size to be in words rather than bytes.
`struct_align`	Layout components of a VAX structure in memory as in VMS Fortran, without padding. Note: this can cause data misalignments.
`%all`	Enable all these VAX VMS features.
`%none`	Disable all these VAX VMS features.

Sub-options can be individually selected or turned off by preceeding with `no%`.

Example:

```
-vax=debug,rsize,no%blank_zero
```

-vpara

Show verbose parallelization messages.

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: Verbose parallelization warnings:

```
demo% f95 -explicitpar -vpara any.f
any.f:
 MAIN any:
"any.f", line 11: Warning: the loop may have parallelization
inhibiting reference
```

-w[*n*]

Show or suppress warning messages.

This option shows or 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.

n may be 0, 1, 2 ,3, or 4.

`-w0` shows just error messages. This is equivalent to `-w`
`-w1` shows errors and warnings. This is the default without `-w`.
`-w2` shows errors, warnings, and cautions.
`-w3` shows errors, warnings, cautions, and notes.
`-w4` shows errors, warnings, cautions, notes, and comments.

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

```
demo% f95 -w -parallel any.f
f95: Warning: Optimizer level changed from 0 to 3 to support
parallelized code
demo%
```

-Xlist[*x*]

Produce listings and do global program checking (GPC).

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 3-9 `-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
`-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
`-XlistMP`	Check OpenMP directives (**SPARC**)
`-Xlisto` *name*	Output report file to *name* instead of *file*`.lst`
`-Xlists`	Suppress unreferenced names from the cross–reference table
`-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

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

-x386

(**x86**) Synonym for `-xtarget=386`

-x486

(**x86**) Synonym for `-xtarget=486`

-xa

Synonym for `-a`.

-xalias[**=***keywords*]

Specify degree of aliasing to be assumed by the compiler.

Some non-standard programming techniques can introduce situations that interfere with the compiler's optimization strategies. The use of overindexing, pointers, and passing global or non-unique variables as subprogram arguments, can introduce ambiguous aliasing situations that could result code that does not work as expected.

Use the `-xalias` flag to inform the compiler about the degree to which the program deviates from the aliasing requirements of the Fortran standard.

The flag may appear with or without a list of keywords. The *keywords* list is comma-separated, and each keyword indicates an aliasing situation present in the program.

Each keyword may be prefixed by `no%` to indicate an aliasing type that is not present.

The aliasing keywords are:

TABLE 3-10 `-xalias` Option Keywords

keyword	meaning
`dummy`	Dummy (formal) subprogram parameters can alias each other and global variables.
`no%dummy`	(Default). Usage of dummy parameters follows the Fortran standard and do not alias each other or global variables.
`craypointer`	(Default) (Default). Cray pointers can point at any global variable or a local variable whose address is taken by the `LOC()` function. Also, two Cray pointers might point at the same data. This is a safe assumption that could inhibit some optimizations.
`no%craypointer`	Cray pointers point only at unique memory addresses, such as obtained from `malloc()`. Also, no two Cray pointers point at the same data. This assumption enables the compiler to optimize Cray pointer references.
`actual`	The compiler treats actual subprogram arguments as if they were global variables. Passing an argument to a subprogram might result in aliasing through Cray pointers.
`no%actual`	(Default) Passing an argument does not result in further aliasing.

TABLE 3-10 `-xalias` Option Keywords *(Continued)*

keyword	meaning
`overindex`	• A reference to an element in a COMMON block might refer to any element in a COMMON block or equivalence group. • Passing any element of a COMMON block or equivalence group as an actual argument to a subprogram gives access to any element of that COMMON block or equivalence group to the called subprogram. • Variables of a sequence derived type are treated as if they were COMMON blocks, and elements of such a variable might alias other elements of that variable. • Individual array bounds may be violated, but except as noted above, the referenced array element is assumed to stay within the array. Array syntax, `WHERE`, and `FORALL` statements are not considered for overindexing. If overindexing occurs in these constructs, they should be rewritten as `DO` loops.
`no%overindex`	(Default) Array bounds are not violated. Array references do not reference other variables.
`ftnpointer`	Calls to external functions might cause Fortran pointers to point at target variables of any type, kind, or rank.
`no%ftnpointer`	(Default) Fortran pointers follow the rules of the standard.

Specifying `-xalias` without a list gives the best performance for most programs that do not violate Fortran aliasing rules, and corresponds to:

```
no%dummy,no%craypointer,no%actual,no%overindex,no%ftnpointer
```

To be effective, `-xalias` should be used when compiling with optimization levels `-xO3` and higher.

The compiler default, with no `-xalias` flag specified, assumes that the program conforms to the Fortran 95 standard except for Cray pointers:

```
no%dummy,craypointer,no%actual,no%overindex,no%ftnpointer
```

Examples of various aliasing situations and how to specify them with `-xalias` are given in the Porting chapter of the *Fortran Programming Guide*.

-xarch=*isa*

Specify instruction set architecture (ISA).

Architectures that are accepted by `-xarch` keyword *isa* are shown in TABLE 3-11:

TABLE 3-11 `-xarch` ISA Keywords

Platform	Valid `-xarch` Keywords
SPARC	`generic`, `generic64`, `native`, `native64`, `v7`, `v8a`, `v8`, `v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, `v9b`
x86	`generic`, `native`, `386`, `pentium_pro`, `sse`, `sse2`, `amd64`

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 3-12 summarizes the most general `-xarch` options on SPARC platforms.

TABLE 3-12 Most General `-xarch` Options on SPARC Platforms

`-xarch=`	Performance
`generic`	• runs adequately on all supported platforms
`v8plusa`	• runs optimally on UltraSPARC-II processors in 32-bit mode
`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.

The default when `-xarch` is not specified is `v8plus` on SPARC platforms, `386` on x86 platforms.

TABLE 3-13 gives details for each of the `-xarch` keywords on SPARC platforms.

TABLE 3-13 `-xarch` Values for SPARC Platforms

`-xarch=`	**Meaning (SPARC)**
`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 `v8plus`.
`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.

TABLE 3-13 `-xarch` Values for SPARC Platforms *(Continued)*

`-xarch=`	**Meaning (SPARC)**
`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
`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

TABLE 3-13 `-xarch` Values for SPARC Platforms *(Continued)*

-xarch=	Meaning (SPARC)
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.
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.

TABLE 3-14 details each of the -xarch keywords on x86 platforms. The default on x86 is generic if -xarch is not specified.

TABLE 3-14 -xarch Values for x86 Platforms

-xarch=	Meaning (x86)
generic	Compile for good performance on most 32-bit x86 platforms. This is the default, and is equivalent to -xarch=386.
generic64	Compile for good performance on most 64-bit x86 platforms. It is interpreted as amd64 currently.
native	Compile for good performance on this x86 architecture. Use the best instruction set for good performance on most x86 processors. With each new release, the definition of "best" instruction set may be adjusted, if appropriate.
native64	Compile for good performance on this 64-bit x86 architecture.
386	Limits instruction set to the Intel 386/486 architecture.
pentium_pro	Limits instruction set to the Pentium Pro architecture.
sse	Adds the SSE instruction set to pentium_pro. (See Note below.)
sse2	Adds the SSE2 instruction set to the pentium_pro. (See Note below.)
amd64	Compile for AMD64 64-bit x86 instruction set.

Cautions on x86 Platforms:

Programs compiled with -xarch={sse|sse2} to run on Solaris x86 SSE/SSE2 Pentium 4-compatible platforms must be run only on platforms that are SSE/SSE2 enabled. Running such programs on platforms that are not SSE/SSE2-enabled could result in segmentation faults or incorrect results occuring without any explicit warning messages. Patches to the OS and compilers to prevent execution of SSE/SSE2-compiled binaries on platforms not SSE/SSE2-enabled might be made available at a later date.

OS releases starting with Solaris 9/04 are SSE/SSE2-enabled on Pentium 4-compatible platforms. Earlier versions of Solaris OS are not SSE/SSE2-enabled. This warning extends also to programs that employ .il inline assembly language functions or __asm() assembler code that utililize SSE/SSE2 instructions.

If you compile and link in separate steps, always link using the compiler and with -xarch={sse|sse2} to ensure that the correct startup routine is linked.

Arithmetic results on x86 may differ from results on SPARC due to the x86 80-bit floating-point registers. To minimize these differences, use the -fstore option or compile with -xarch=sse2 if the hardware supports SSE2.

-xassume_control[=*keywords*]

Set parameters to control ASSUME pragmas.

Use this flag to control the way the compiler handles ASSUME pragmas in the source code.

The ASSUME pragmas provide a way for the programmer to assert special information that the compiler can use for better optimization. These assertions may be qualified with a probability value. Those with a probability of 0 or 1 are marked as certain; otherwise they are considered non-certain.

You can also assert, with a probability or certainty, the trip count of an upcoming DO loop, or that an upcoming branch will be taken.

See Section 2.3.1.9, "The ASSUME Directives" on page 2-13, for a description of the ASSUME pragmas recognized by the f95 compiler.

The *keywords* on the -xassume_control option can be a single suboption keyword or a comma-separated list of keywords. The keyword suboptions recognized are:

optimize	The assertions made on ASSUME pragmas affect optimization of the program.
check	The compiler generates code to check the correctness of all assertions marked as certain, and emits a runtime message if the assertion is violated; the program continues if fatal is not also specified.
fatal	When used with check, the program will terminate when an assertion marked certain is violated.
retrospective[:*d*]	The *d* parameter is an optional tolerance value, and must be a real positive constant less than 1. The default is ".1". retrospective compiles code to count the truth or falsity of all assertions. Those outside the tolerance value *d* are listed on output at program termination.
%none	All ASSUME pragmas are ignored.

The compiler default is

```
-xassume_control=optimize
```

This means that the compiler recognizes ASSUME pragmas and they will affect optimization, but no checking is done.

If specified without parameters, -xassume_control implies

```
-xassume_control=check,fatal
```

In this case the compiler accepts and checks all certain `ASSUME` pragmas, but they do not affect optimization. Assertions that are invalid cause the program to terminate.

-xautopar

Synonym for `-autopar`.

-xcache=*c*

Define cache properties for the optimizer.

c must be one of the following:

- `generic`
- native
- *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 processors without any major performance degradation. This is the default.
native	Define the cache properties for good performance on this host platform.
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

(SPARC) Synonym for `-cg89`.

-xcg92

(SPARC) Synonym for `-cg92`.

-xcheck=*keyword*

Generate special runtime checks and initializations.

The *keyword* must be one of the following:

keyword	**Feature**
`stkovf`	Turn on runtime checking for stack overflow on subprogram entry. If a stack overflow is detected, a `SIGSEGV` segment fault will be raised. (**SPARC** only)
`no%stkovf`	Disable runtime checking for stack overflow. (**SPARC** only)
init_local	Perform special initialization of local variables. The compiler initializes local variables to a value that is likely to cause an arithmetic exception if it is used by the program before it is assigned. Memory allocated by the `ALLOCATE` statement will also be initialized in this manner. Module variables, `SAVE` variables, and variables in `COMMON` blocks are not initialized.
`no%init_local`	Disable local variable initialization. This is the default.
`%all`	Turn on all these runtime checking features.
`%none`	Disable all these runtime checking features.

Stack overflows, especially in multithreaded applications with large arrays allocated on the stack, can cause silent data corruption in neighboring thread stacks. Compile all routines with `-xcheck=stkovf` if stack overflow is suspected. But note that compiling with this flag does not guarantee that all stack overflow situations will be detected since they could occur in routines not compiled with this flag.

-xchip=*c*

Specify target processor for the optimizer.

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 tables list the valid `-xchip` processor name values:

TABLE 3-16 Common `-xchip` SPARC Processor Names

-xchip=	Optimize for:
`generic`	most SPARC processors. (This is the default.)
`native`	this host platform.
`ultra`	the UltraSPARC processor.
`ultra2`	the UltraSPARC II processor.
`ultra2e`	the UltraSPARC IIe processor.
`ultra2i`	the UltraSPARC IIi processor.
`ultra3`	the UltraSPARC III processor.
`ultra3cu`	the UltraSPARC IIIcu processor.
`ultra4`	the UltraSPARC IV processor

The following are older, less common `-xchip` processor names and are listed here for reference purposes only:

TABLE 3-17 Less Common `-xchip` SPARC Processor Names

-xchip=	Optimize for:
`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.

On x86 platforms: the `-xchip` values are `386`, `486`, `pentium`, `pentium_pro`, `pentium3`, `pentium4`, `opteron`, `generic`, and `native`.

`-xcode=`*keyword*

(SPARC) Specify code address space on SPARC platforms.

The values for *keyword* are:

keyword	**Feature**
`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=`*keyword* explicitly) are:

`-xcode=abs32` on SPARC V8 and V7 platforms.
`-xcode=abs44` on UltraSPARC V9 (`-xarch=v9` platforms)

Position-Independent Code:

Use `-xcode=pic13` or `-xcode=pic32` when creating dynamic shared libraries to improve runtime performance.

While the code within a dynamic executable is usually tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of the process.

When you use position-independent code, relocatable references are generated as an indirect reference through a global offset table. Frequently accessed items in a shared object will benefit from compiling with `-xcode=pic13` or `-xcode=pic32` by not requiring the large number of relocations imposed by code that is not position-independent.

The size of the global offset table is limited to 8Kb.

There are two nominal performance costs with -xcode={pic13|pic32} :

- A routine compiled with either -xcode=pic13 or -xcode=pic32 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 pic32, there are two additional instructions per global and static memory reference.

When considering the above costs, remember that the use of -xcode=pic13 or -xcode=pic32 can significantly reduce system memory requirements, due to the effect of library code sharing. Every page of code in a shared library compiled -xcode=pic13 or -xcode=pic32 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 -xcode=pic13 or -xcode=pic32 is with the nm command:

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

A .o file containing position-independent code will contain an unresolved external reference to _GLOBAL_OFFSET_TABLE_ as marked by the letter U.

To determine whether to use -xcode=pic13 or -xcode=pic32, check the size of the Global Offset Table (GOT) by using elfdump -c (see the elfdump(1) man page for more information) and for the section header, sh_name: .got. The sh_size value is the size of the GOT. If the GOT is less than 8,192 bytes, specify -xcode=pic13, otherwise specify -xcode=pic32.

In general, use the following guidelines to determine how you should use -xcode:

- If you are building an executable you should not use -xcode=pic13 or -xcode=pic32.
- If you are building an archive library only for linking into executables you should not use -xcode=pic13 or -xcode=pic32.
- If you are building a shared library, start with -xcode=pic13 and once the GOT size exceed 8,192 bytes, use -xcode=pic32.
- If you are building an archive library for linking into shared libraries you should just use -xcode=pic32.

Compiling with the -xcode=pic13 or pic32 (or -pic or -PIC) options is recommended when building dynamic libraries. See the Solaris Linker and Libraries Guide.

-xcommonchk[={yes|no}]

Enable runtime checking of common block inconsistencies.

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 -xcommonchk=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. -xcommonchk without a value is equivalent to -xcommonchk=yes.

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[={1|0}]

Enable optimization and inlining across source files.

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 Section , "-inline=[%auto][[,][no%]f1,...[no%]fn]" on page 3-32.

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).

Any .s assmbler source files in the compilation do not participate in the crossfile analysis. Also, the -xcrossfile flag is ignored if compiling with -S.

-xdebugformat={stabs|dwarf}

Sun Studio compilers are migrating the format of debugger information from the "stabs" format to the "dwarf" format. The default setting for this release is -xdebugformat=stabs.

If you maintain software which reads debugging information, you now have the option to transition your tools from the stabs format to the dwarf format.

Use this option as a way of accessing the new format for the purpose of porting tools. There is no need to use this option unless you maintain software which reads debugger information, or unless a specific tool tells you that it requires debugger information in one of these formats.

-xdebugformat=stabs generates debugging information using the stabs standard format.

-xdebugformat=dwarf generates debugging information using the dwarf standard format.

If you do not specify -xdebugformat, the compiler assumes -xdebugformat=stabs. It is an error to specify the option without an argument.

This option affects the format of the data that is recorded with the `-g` option. Some the format of that information is also controlled with this option. So `-xdebugformat` has a an effect even when `-g` is not used.

The dbx and Performance Analyzer software understand both stabs and dwarf format so using this option does not have any effect on the functionality of either tool.

This is a transitional interface so expect it to change in incompatible ways from release to release, even in a minor release. The details of any specific fields or values in either stabs or dwarf are also evolving.

-xdepend

Synonym for `-depend`.

-xexplicitpar

Synonym for `-explicitpar`.

-xF

Allow function–level reordering by the Performance Analyzer.

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. See the *Program Performance Analysis Tools* manual for further information on the analyzer.

-xfilebyteorder=*options*

Support file sharing between little-endian and big-endian platforms.

The flag identifies the byte-order and byte-alignment of data on unformatted I/O files. *options* must specify any combination of the following, but at least one specification must be present:

little*max_align***:***spec*

big*max_align***:***spec*

native:*spec*

max_align declares the maximum byte alignment for the target platform. Permitted values are 1, 2, 4, 8, and 16. The alignment applies to Fortran VAX structures and Fortran 95 derived types that use platform-dependent alignments for compatibility with C language structures.

little specifies a "little-endian" file on platforms where the maximum byte alignment is *max_align*. For example, `little4` specifies a 32-bit x86 file, while `little16` describes a 64-bit x86 file.

big specifies a "big-endian" file with a maximum alignment of *max_align*. For example, `big8` describes a SPARCV8 (32-bit) file, while `big16` describesa SPARC V9 (64-bit) file.

native specifies a "native" file with the same byte order and alignment used by the compiling processor platform. The following are assumed to be "native":

Platform	"native" corresponds to:
32-bit SPARC V8	`big8`
64-bit SPARC V9	`big16`
32-bit x86	`little4`
64-bit x86 (`amd64`)	`little16`

spec must be a comma-separated list of the following:

%all

unit

filename

%all refers to all files and logical units except those opened as `"SCRATCH"`, or named explicitly elsewhere in the `-xfilebyteorder` flag. `%all` can only appear once.

unit refers to a specific Fortran unit number opened by the program.

filename refers to a specific Fortran file name opened by the program.

Examples:

```
-xfilebyteorder=little4:1,2,afile.in,big8:9,bfile.out,12
-xfilebyteorder=little8:%all,big16:20
```

Notes:

This option does not apply to files opened with `STATUS="SCRATCH"`. I/O operations done on these files are always with the byte-order and byte-alignment of the native processor.

The first default, when `-xfilebyteorder` does not appear on the command line, is `-xfilebyteorder=native:%all`.

A file name or unit number can be declared only once in this option.

When `-xfilebyteorder` does appear on the command line, it must appear with at least one of the little, big, or native specifications.

Files not explicitly declared by this flag are assumed to be native files. For example, compiling with `-xfilebyteorder=little4:zork.out` declares `zork.out` to be a little-endian 32-bit x86 file with a 4-byte maximum data alignment. All other files in the program are native files.

When the byte-order specified for a file is the same as the native processor but a different alignment is specified, the appropriate padding will be used even though no byte swapping is done. For example,this would be the case when compiling with `-xarch=amd64` for 64-bit x86 platforms and `-xfilebyteorder=little4:`*filename* is specified.

The declared types in data records shared between big-endian and little-endian platforms must have the same sizes. For example, a file produced by a SPARC executable compiled with `-xtypemap=integer:64,real:64,double:128` cannot be read by an x86 executable compiled with `-xtypemap=integer:64,real:64,double:64` since the default double precision datatypes will have different sizes.

Note that unformatted files containing `REAL*16` data cannot be used on x86 platforms, which do not support `REAL*16`.

An I/O operation withan entire `UNION/MAP` data object on a file specified as non-native will result in a runtime I/O error. You can only execute I/O operations using the individual members of the `MAP` (and not an entire VAX record containing the `UNION/MAP`) on non-native files.

-xhasc[={yes|no}]

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

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 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% f95 -o has0 hasc.f
demo% has0
 string length =   4     <-- should be 7
demo% f95 -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 legacy Fortran 77 programs.

-xhelp={readme|flags}

Show summary help information.

-xhelp=readme Show the online README file for this release of the compiler.
-xhelp=flags List the compiler option flags. Equivalent to -help.

-xia[={widestneed|strict}]

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

The default if not specified is -xia=widestneed.

Fortran 95 extensions for interval arithmetic calculations are detailed in the *Interval Arithmetic Programming Reference.* See also Section , "-xinterval[={widestneed | strict | no}]" on page 3-73.

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, ild.

-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. With previous releases of the compilers, -xildon was the default when compiling only object files with -g. To force the use of ild in link-only compiles with -g, include the -xildon option explicitly.

See Section , "-g" on page 3-30, and the section on ild in the *C User's Guide*.

-xinline=*list*

Synonym for -inline.

-xinterval[={*widestneed* | *strict* | *no*}]

(SPARC) Enable interval arithmetic extensions.

The optional value 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 *Fortran 95 Interval Arithmetic Programming Reference.* See also Section , "-xia[={widestneed | strict}]" on page 3-72.

-xipo[={0 | 1 | 2}]

(SPARC) Perform interprocedural optimizations.

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. `-xipo=2` adds interprocedural aliasing analysis and memory allocation and layout optimizations to improve cache performance. The default is `-xipo=0`, and if `-xipo` is specified without a value, `-xipo=1` is used.

When compiling with `-xipo=2`, there should be no calls from functions or subroutines compiled without `-xipo=2` (for example, from libraries) to functions or subroutines compiled with `-xipo=2`.

As an example, if you interpose on the function `malloc()` and compile your own version of `malloc()` with `-xipo=2`, all the functions that reference `malloc()` in any library linked with your code would also have to be compiled with `-xipo=2`. Since this might not be possible for system libraries, your version of `malloc` should not be compiled with `-xipo=2`.

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`.
- `.s` assembly language files do not participate in interprocedural analysis.
- The `-xipo` flag is ignored if compiling with `-S`.

When Not To Compile With `-xipo`:

Working with the set of object files in the link step, the compiler tries to perform whole-program analysis and optimizations. For any function or subroutine `foo()` defined in this set of object files, the compiler makes the following two assumptions:

(1) At runtime, `foo()` will not be called explicitly by another routine defined outside this set of object files, and

(2) calls to `foo()` from any routine in the set of object files will be not be interposed upon by a different version of `foo()` defined outside this set of object files.

If assumption (1) is not true for the given application, do not compile with `-xipo=2`.
If assumption (2) is not true, do not compile with either `-xipo=1` or `-xipo=2`.

As an example, consider interposing on the function `malloc()` with your own source version and compiling with `-xipo=2`. Then all the functions in any library that reference `malloc()` that are linked with your code would have to also be compiled with `-xipo=2` and their object files would need to participate in the link step. Since this might not be possible for system libraries, your version of malloc should not be compiled with `-xipo=2`.

As another example, suppose that you build a shared library with two external calls, `foo()` and `bar()` inside two different source files, and `bar()` calls `foo()` inside its body. If there is a possibility that the function call `foo()` could be interposed at runtime, then compile neither source file for `foo()` or `bar()` with `-xipo=1` or `-xipo=2`. Otherwise, `foo()` could be inlined into `bar()`, which could cause incorrect results when compiled with `-xipo`.

`-xipo_archive[={none|readonly|writeback}]`

(SPARC) Allow crossfile optimization to include archive (`.a`) libraries.

The value must be one of the following:

`writeback`	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the `.a` archive library before producing an executable. Any object files contained in the library that were optimized during the compilation are replaced with their optimized version.
`readonly`	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the `.a` archive library before producing an executable.
`none`	No processing of archive files is performed.

If you do not specify a setting for `-xipo_archive`, the compiler assumes `-xipo_archive=none`.

`-xjobs=`*n*

Compile with multiple processors.

Specify the `-xjobs` option to set how many processes the compiler creates to complete its work. This option can reduce the build time on a multi-cpu machine. In this release of the `f95` compiler, `-xjobs` works only with the `-xipo` option. When you specify `-xjobs=`*n*, the interprocedural optimizer uses *n* as the maximum number of code generator instances it can invoke to compile different files.

Generally, a safe value for *n* is 1.5 multiplied by the number of available processors. Using a value that is many times the number of available processors can degrade performance because of context switching overheads among spawned jobs. Also, using a very high number can exhaust the limits of system resources such as swap space.

You must always specify `-xjobs` with a value. Otherwise an error diagnostic is issued and compilation aborts.

Multiple instances of `-xjobs` on the command line override each other until the rightmost instance is reached.

The following example compiles more quickly on a system with two processors than the same command without the `-xjobs` option.

```
example% f95 -xipo -xO4 -xjobs=3 t1.f t2.f t3.f
```

-xknown_lib=*library_list*

Recognize calls to a known library.

When specified, the compiler treats references to certain known libraries as intrinsics, ignoring any user-supplied versions. This enables the compiler to perform optimizations over calls to library routines based on its special knowledge of that library.

The *library_list* is a comma-delimited list of keywords currently to `blas`, `blas1`, `blas2`, `blas3`, and `intrinsics`. The compiler recognizes calls to the following BLAS1, BLAS2, and BLAS3 library routines and is free to optimize appropriately for the Sun Performance Library implementation. The compiler will ignore user-supplied versions of these library routines and link to the BLAS routines in the Sun Performance Library.

-xknown_lib=	Feature
blas1	The compiler recognizes calls to the following BLAS1 library routines: caxpy ccopy cdotc cdotu crotg cscal csrot csscal cswap dasum daxpy dcopy ddot drot drotg drotm drotmg dscal dsdot dswap dnrm2 dzasum dznrm2 icamax idamax isamax izamax sasum saxpy scasum scnrm2 scopy sdot sdsdot snrm2 srot srotg srotm srotmg sscal sswap zaxpy zcopy zdotc zdotu zdrot zdscal zrotg zscal zswap
blas2	The compiler recognizes calls to the following BLAS2 library routines: cgemv cgerc cgeru ctrmv ctrsv dgemv dger dsymv dsyr dsyr2 dtrmv dtrsv sgemv sger ssymv ssyr ssyr2 strmv strsv zgemv zgerc zgeru ztrmv ztrsv
blas3	The compiler recognizes calls to the following BLAS2 library routines: cgemm csymm csyr2k csyrk ctrmm ctrsm dgemm dsymm dsyr2k dsyrk dtrmm dtrsm sgemm ssymm ssyr2k ssyrk strmm strsm zgemm zsymm zsyr2k zsyrk ztrmm ztrsm
blas	Selects all the BLAS routines. Equivalent to `-xknown_lib=blas1,blas2,blas3`
intrinsics	The compiler ignores any explicit EXTERNAL declarations for Fortran 95 intrinsics, thereby ignoring any user-supplied intrinsic routines.

-xlang=f77

(SPARC) Prepare for linking with runtime libraries compiled with earlier versions of f77.

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

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

Note the following when compiling with -xlang:

- Do not compile with both -xnolib and -xlang.

- When mixing Fortran object files with C++, link using the C++ compiler and specify `-xlang=f95` on the `CC` command line.
- When mixing C++ objects with Fortran object files compiled with any of the parallelization options, the linking `CC` command line must also specify `-mt`.

-xlibmil

Synonym for `-libmil`.

-xlibmopt

Use library of optimized math routines.

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.

For example:

```
f95 -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 information.

Use this option to return serial number entitlement information about the installed compiler software.

-xlinkopt[={1|2|0}]

(SPARC) Perform link-time optimizations on relocatable object files.

The post-optimizer performs a number of advanced performance optimizations on the binary object code at link-time. The optional value sets the level of optimizations performed, and must be 0, 1, or 2.

`0` The post-optimizer is disabled. (This is the default.)

`1` Perform optimizations based on control flow analysis, including instruction cache coloring and branch optimizations, at link time.

`2` Perform additional data flow analysis, including dead-code elimination and address computation simplification, at link time.

Specifying the `-xlinkopt` flag without a value implies `-xlinkopt=1`.

These optimizations are performed at link time by analyzing the object binary code. The object files are not rewritten but the resulting executable code may differ from the original object codes.

This option is most effective when used to compile the whole program, and with profile feedback.

When compiling in separate steps, `-xlinkopt` must appear on both compile and link steps.

```
demo% f95 -c -xlinkopt a.f95 b.f95
demo% f95 -o myprog -xlinkopt=2 a.o b.o
```

Note that the level parameter is only used when the compiler is linking. In the example above, the postoptimization level used is 2 even though the object binaries were compiled with an implied level of 1.

The link-time post-optimizer cannot be used with the incremental linker, `ild`. The `-xlinkopt` flag will set the default linker to be `ld`. Enabling the incremental linker explicitly withthe `-xildon` flag will disable the `-xlinkopt` option if both are specified together.

For the `-xlinkopt` option to be useful, at least some, but not necessarily all, of the routines in the program must be compiled with this option. The optimizer can still perform some limited optimizations on object binaries not compiled with `-xlinkopt`.

The `-xlinkopt` option will optimize code coming from static libraries that appear on the compiler command line, but it will skip and not optimize code coming from shared (dynamic) libraries that appear on the command line. You can also use `-xlinkopt` when building shared libraries (compiling with `-G`).

The link-time post-optimizer is most effective when used with run-time profile feedback. Profiling reveals the most and least used parts of the code and directs the optimizer to focus its effort accordingly. This is particularly important with large applications where optimal placement of code performed at link time can reduce instruction cache misses. Typically, this would be compiled as shown below:

```
demo% f95 -o progt -xO5 -xprofile=collect:prog file.f95
demo% progt
demo% f95 -o prog -xO5 -xprofile=use:prog -xlinkopt file.95
```

For details on using profile feedback, see the `-xprofile` option

Note that compiling with this option will increase link time slightly. Object file sizes also increase, but the size of the executable remains the same. Compiling with the `-xlinkopt` and `-g` flags increases the size of the excutable by including debugging information.

-xloopinfo

Synonym for `-loopinfo`.

-xmaxopt[=*n*]

Enable optimization pragma and set maximum optimization level.

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 when it appears in the source input. Without this option, the compiler treats these lines as comments. See Section 2.3.1.5, "The `OPT` Directive" on page 2-12.

If this pragma 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*>]

(SPARC) Specify maximum assumed memory alignment and behavior of misaligned data accesses.

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 when compiling without -xmemalign specified are:

- 8i for -xarch=generic,v7,v8,v8a,v8plus,v8plusa
- 8s for -xarch=v9,v9a with C and C++
- 8f for -xarch=v9,v9a with Fortran

The default for -xmemalign appearing without a value is 1i for all platforms.

Note that -xmemalign itself does not force any particular data alignment to take place. Use -dalign or -aligncommon to force data alignment.

The -dalign option is a macro:

-dalign is a macro for: -xmemalign=8s -aligncommon=16

See Section , "-aligncommon[={1|2|4|8|16}]" on page 3-11 for details.

-xnolib

Synonym for -nolib.

-xnolibmil

Synonym for -nolibmil.

-xnolibmopt

Do not use fast math library.

Use with -fast to override linking the optimized math library:

```
f95 -fast -xnolibmopt …
```

-xO*n*

Synonym for -O*n*.

-xopenmp

(SPARC) Synonym for -openmp.

-xpad

Synonym for -pad.

-xpagesize=*size*

(SPARC) Set the preferred page size for the stack and the heap.

The *size* value must be one of the following:

8K 64K 512K 4M 32M 256M 2G 16G or default

For example: -xpagesize=4M

Not all these page sizes are supported on all platforms and depend on the architecture and Solaris environment. The page size specified must be a valid page size for the Solaris operating environment on the target platform, as returned by getpagesizes(3C). If it is not, the request will be silently ignored at run-time. The Solaris environment offers no guarantee that the page size request will be honored.

You can use pmap(1) or meminfo(2) to determine if your running program received the requested page size.

If you specify -xpagesize=default, the flag is ignored; -xpagesize specified without a *size* value is equivalent to -xpagesize=default.

This option is a macro for

-xpagesize_heap=*size* -xpagesize_stack=*size*

These two options accept the same arguments as -xpagesize: 8K, 64K, 512K, 4M, 32M, 256M, 2G, 16G, default. You can set them both with the same value by specifying -xpagesize=*size* or you can specify them individually with different values.

Compiling with this flag has the same effect as setting the LD_PRELOAD environment variable to mpss.so.1 with the equivalent options, or running the Solaris 9 command ppgsz(1) with the equivalent options, before starting the program. See the Solaris 9 man pages for details.

Note that this feature is not available on Solaris 7 and 8 environments. A program compiled with this option will not link on Solaris 7 and 8 environments.

-xpagesize_heap=*size*

(SPARC) Set the preferred page size for the heap.

The *size* value must be one of the following:

`8K 64K 512K 4M 32M 256M 2G 16G` or `default`

For example: `-xpagesize_heap=4M`

See `-xpagesize` for details.

-xpagesize_stack=*size*

(SPARC) Set the preferred page size for the stack.

The *size* value must be one of the following:

`8K 64K 512K 4M 32M 256M 2G 16G` or `default`

For example: `-xpagesize_stack=4M`

See `-xpagesize` for details.

-xparallel

Synonym for `-parallel`.

-xpg

Synonym for `-pg`.

-xpp={fpp | cpp}

Select source file preprocessor.

The default is `-xpp=fpp`.

The compilers use `fpp`(1) to preprocess `.F`, `.F95`, or `.F03` 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 those architectures that support prefetch, such as UltraSPARC II or UltraSPARC III, Pentium 3, Pentium 4, or AMD Opteron (`-xarch=v8plus`, `v8plusa`, `v9plusb`, `v9`, `v9a`, or `v9b`, `sse`, `sse2`, `generic64`, or `amd64`)

See Section 2.3.1.8, "The `PREFETCH` Directives" on page 2-13 for a description of the Fortran `PREFETCH` directives.

a must be one of the following:

a is	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 `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.

-xprefetch_auto_type=[no%]indirect_array_access

Generate indirect prefetches for a data arrays accessed indirectly.

Does [not] generate indirect prefetches for the loops indicated by the option `-xprefetch_level={1|2|3}` in the same fashion the prefetches for direct memory accesses are generated. The prefix `no%` negates the declaration.

If you do not specify a setting for `-xprefetch_auto_type`, the compiler sets it to `-xprefetch_auto_type=no%indirect_array_access`.

Requires `-xprefetch=auto` and an optimization level `-xO3` or higher.

Options such as `-xdepend` can affect the aggressiveness of computing the indirect prefetch candidates and therefore the aggressiveness of the automatic indirect prefetch insertion due to better memory alias disambiguation information.

-xprefetch_level={1|2|3}

Control the automatic generation of prefetch instructions.

This option is only effective when compiling with:

- `-xprefetch=auto`,
- with optimization level 3 or greater,
- on a platform that supports prefetch (`-xarch=v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, `v9b`, `generic64`, `native64`).

The default for `-xprefetch=auto` without specifying `-xprefetch_level` is level 2.

Prefetch level 2 generates additional opportunities for prefetch instructions than level 1. Prefetch level 3 generates additional prefetch instructions than level 2.

Prefetch levels 2 and 3 are only effective on UltraSPARC III platforms (`-xarch=v8plusb` or `v9b`), or x86 Pentium 4 or AMD Opteron (`-xarch=sse2` or `amd64`)

-xprofile={**collect**[**:***name*]|**use**[**:***name*]|**tcov**}

Collect or optimize with runtime profiling data, or perform basic block coverage analysis.

Compiling with high optimization levels (`-xO5`) is enhanced by providing the compiler with runtime performance feedback. To produce the profile feedback the compiler needs to do its best optimizations, you must compile first with `-xprofile=collect`, run the executable against a typical data set, and then recompile at the highest optimization level and with `-xprofile=use`.

`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 named `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.

Profile collection is “MT-safe”. That is, profiling a program that does its own multitasking by compiling with `-mt` and calling the multitasking library directly will give accurate results.

When compiling and linking in separate steps, the link step must also specify `-xprofile=collect` if it appears on the compile step.

`use`[`:`*nm*]

Use execution frequency data to optimize strategically at optimization level `-xO5`.

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 profile data 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*.

See also `-xprofile_ircache` for speeding up compilationg between the collect and use phases.

See also `-xprofile_pathmap` for controlling where the compiler looks for profile data files.

`tcov`

Basic block coverage analysis using "new" style `tcov`. Optimization level must be `-O2` or greater.

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 *Program Performance Analysis Tools* 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.

`-xprofile_ircache`[`=`*path*]

(SPARC) Save and reuse compilation data between collect and use profile phases.

Use with `-xprofile=collect|use` to improve compilation time during the use phase by reusing compilation data saved from the collect phase.

If specified, *path* will override the location where the cached files are saved. By default, these files will be saved in the same directory as the object file. Specifying a path is useful when the collect and use phases happen in two different places.

A typical sequence of commands might be:

```
demo% f95 -xO5 -xprofile=collect -xprofile_ircache t1.c t2.c
demo% a.out      collects feedback data
demo% f95 -xO5 -xprofile=use -xprofile_ircache t1.c t2.c
```

With large programs, compilation time in the use phase can improve significantly by saving the intermediate data in this manner. But this will be at the expense of disk space, which could increase considerably.

-xprofile_pathmap=*collect_prefix*:*use_prefix*

(SPARC) Set path mapping for profile data files.

Use the `-xprofile_pathmap` option with the `-xprofile=use` option.

Use `-xprofile_pathmap` when the compiler is unable to find profile data for an object file that is compiled with `-xprofile=use`, and:

- You are compiling with `-xprofile=use` into a directory that is not the directory used when previously compiling with `-xprofile=collect`.
- Your object files share a common basename in the profile but are distinguished from each other by their location in different directories.

The *collect-prefix* is the prefix of the UNIX pathname of a directory tree in which object files were compiled using `-xprofile=collect`.

The *use-prefix* is the prefix of the UNIX pathname of a directory tree in which object files are to be compiled using `-xprofile=use`.

If you specify multiple instances of `-xprofile_pathmap`, the compiler processes them in the order of their occurrence. Each *use-prefix* specified by an instance of `-xprofile_pathmap` is compared with the object file pathname until either a matching *use-prefix* is identified or the last specified *use-prefix* is found not to match the object file pathname.

-xrecursive

Allow routines without `RECURSIVE` attribute call themselves recursively.

Normally, only subprograms defined with the `RECURSIVE` attribute can call themselves recursively.

Compiling with `-xrecursive` enables subprograms to call themselves, even if they are not defined with the `RECURSIVE` attribute. But, unlike subroutines defined `RECURSIVE`, use of this flag does not cause local variables to be allocated on the

stack by default. For local variables to have separate values in each recursive invocation of the subprogram, compile also with `-stackvar` to put local variables on the stack.

Indirect recursion (routine A calls routine B which then calls routine A) can give inconsistent results at optimization levels greater than `-xO2`. Compiling with the `-xrecursive` flag guarantees correctness with indirect recursion, even at higher optimization levels.

Compiling with `-xrecursive` can cause performance degradations.

-xreduction

Synonym for `-reduction`.

-xregs=*r*

(SPARC) Specify register usage.

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 the compiler to use the application registers as scratch 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=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 the compiler to use the floating–point registers as scratch registers for integer values. This option has no effect on the compiler's use of floating-point registers for floating-point values.
- `no%float`: Do not use the floating–point registers. With this option, a source program cannot contain any floating–point code.

The compiler default is: `-xregs=appl,float`.

-xs

Allow debugging by `dbx` without object (`.o`) files.

With `-xs`, all debug information is copied into the executable file. If you move executables to another directory, then you can use `dbx` and ignore the object (`.o`) files. Use this option when you cannot retain the `.o` files.

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.

-xsafe=mem

(SPARC) Allow the compiler to assume that no memory protection violations occur.

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`

Caution – 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

(Obsolete) Synonym for `-sb`.

-xsbfast

(Obsolete) Synonym for `-sbfast`.

-xspace

Do no optimizations that increase the code size.

Example: Do not unroll or parallelize loops if it increases code size.

-xtarget=*t*

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.

SPARC Platforms

Use the `fpversion`(1) command to determine the expansion of `-xtarget=native` on a running system.

Note that `-xtarget` for a specific host platform might not expand to the same `-xarch`, `-xchip`, or `-xcache` settings as `-xtarget=native` when compiling on that platform.

The following table gives a list of the commonly used system platform names accepted by the compiler. Appendix C gives a list of older and less commonly used system platform names

TABLE 3-18 Expansions of Commonly Used `-xtarget` System Platforms

-xtarget=*platform-name*	**-xarch**	**-xchip**	**-xcache**
`generic`	`generic`	`generic`	`generic`
`generic64`	`v9`	`generic`	`generic`
`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`
`entr5000`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr6000`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`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`

TABLE 3-18 Expansions of Commonly Used `-xtarget` System Platforms *(Continued)*

-xtarget=*platform-name*	**-xarch**	**-xchip**	**-xcache**
`ultra3cu`	`v8plusa`	`ultra3cu`	`64/32/4:8192/512/2`
`ultra3i`	`v8plusa`	`ultra3i`	`64/32/4:1024/64/4`
`ultra4`	`v8plusa`	`ultra4`	`64/32/4:8192/128/2`

Compiling for a 64-bit Solaris OS on UltraSPARC V9 platforms is indicated by the `-xarch=v9` or `-xarch=v9a` flag. Setting `-xtarget=ultra` or `ultra2` is not necessary or sufficient. If `-xtarget` is specified, the `-xarch=v9` or `v9a` option must appear after the `-xtarget` flag, as in:

```
-xtarget=ultra2 ...-xarch=v9
```

otherwise the `-xtarget` setting will revert `-xarch` to `v8plusa`.

x86 Platforms

The valid `-xtarget` platform names for x86 systems are:

`generic, native, 386, 486, pentium, pentium_pro, pentium3, pentium4`, and `opteron.`

Compiling for 64-bit Solaris OS on 64-bit x86 AMD Opteron platforms is indicated by the `-xarch=amd64` flag. Compiling with `-xtarget=opteron` is not necessary or sufficient. If `-xtarget` is specified, the `-xarch=amd64` option must appear after the `-xtarget` flag, as in:

```
-xtarget=opteron -xarch=amd64
```

otherwise the compilation will revert to 32-bit x86.

-xtime

Synonym for `-time`.

-xtypemap=*spec*

Specify default data mappings.

This option provides a flexible way to specify the byte sizes for default data types. This option 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 allowable combinations on each platform are:

- real:32
- real:64
- double:64
- double:128
- integer:32
- integer:64

For example:

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

maps both default REAL and DOUBLE to 8 bytes.

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.

Note that INTEGER and LOGICAL are treated the same, and COMPLEX is mapped as two REALs. Also, DOUBLE COMPLEX will be treated the way DOUBLE is mapped.

-xunroll=*n*

Synonym for -unroll=*n*.

-xvector[={yes|no}]

Enable automatic calls to vectorized library functions.

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.

-ztext

Generate only pure libraries with no relocations.

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% f95 -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% f95 -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.

CHAPTER 4

Fortran 95 Features and Differences

This appendix shows some of the major features differences between standard Fortran 95 and the Fortran 95 compiler, `f95`.

4.1 Source Language Features

The Fortran 95 compiler provides the following source langauge features and extensions to the Fortran 95 standard.

4.1.1 Continuation Line Limits

`f95` allows 99 continuation lines (1 initial and 99 continuation lines). Standard Fortran 95 allows 19 for fixed-form and 39 for free-form.

4.1.2 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.

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.
```

4.1.3 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 4-1 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.

4.1.3.1 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.

4.1.3.2 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.

4.1.4 Limits and Defaults

- A single Fortran 95 program unit can define up to 65,535 derived types and 16,777,215 distinct constants.
- Names of variables and other objects can be up to 127 characters long. 31 is standard.

4.2 Data Types

This section describes features and extensions to the Fortran 95 data types.

4.2.1 Boolean Type

`f95` supports constants and expressions of Boolean type. However, there are no Boolean variables or arrays, and there is no Boolean type statement.

4.2.1.1 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.

4.2.1.2 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

*ddddddd*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:

```
nH...          '...'H          "..."H
nL...          '...'L          "..."L
nR...          '...'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.

4.2.1.3 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.

4.2.2 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 4-2 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 four-byte floating-point
`REAL*8`	`REAL(KIND=8)`	`REAL(8)`	IEEE double-precision eight-byte floating-point
`REAL*16`	`REAL(KIND=16)`	`REAL(16)`	IEEE quad-precision sixteen-byte floating-point

TABLE 4-2 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)

4.2.3 Size and Alignment of Data Types

Storage and alignment are always given in bytes. Values that can fit into a single byte are byte-aligned.

The size and alignment of types depends on various compiler options and platforms, and how variables are declared. The default maximum alignment in COMMON blocks is to 4-byte boundaries.

Default data alignment and storage allocation can be changed by compiling with special options, such as `-aligncommon`, `-f`, `-dalign`, `-dbl_align_all`, `-xmemalign`,, and `-xtypemap`. The default descriptions in this manual assume that these options are not in force.

There is additional information in Chapter 11 of the *Fortran Programming Guide* regarding special cases of data types and alignment on certain platforms.

The following table summarizes the default size and alignment, ignoring other aspects of types and options.

TABLE 4-3 Default Data Sizes and Alignments (in Bytes)

Fortran 95 Data Type	Size	Default Alignment	Alignment in COMMON
BYTE X	1	1	1
CHARACTER X	1	1	1
CHARACTER*n X	n	1	1
COMPLEX X	8	4	4
COMPLEX*8 X	8	4	4
DOUBLE COMPLEX X	16	8	4
COMPLEX*16 X	16	8	4
COMPLEX*32 X	32	8/16	4
DOUBLE PRECISION X	8	8	4
REAL X	4	4	4
REAL*4 X	4	4	4
REAL*8 X	8	8	4
REAL*16 X	16	8/16	4
INTEGER X	4	4	4
INTEGER*2 X	2	2	2
INTEGER*4 X	4	4	4
INTEGER*8 X	8	8	4
LOGICAL X	4	4	4
LOGICAL*1 X	1	1	1
LOGICAL*2 X	2	2	2
LOGICAL*4 X	4	4	4
LOGICAL*8 X	8	8	4

Note the following:

- `REAL*16` and `COMPLEX*32`: in 64-bit environments (compiling with `-xarch=v9` or `v9a`) the default alignment is on 16-byte (rather than 8-byte) boundaries, as indicated by 8/16 in the table. This data type, "quad precision", is not available on x86 platforms.
- Arrays and structures align according to their elements or fields. An array aligns the same as the array element. A structure aligns the same as the field with the widest alignment.

Options `-f` or `-dalign` force alignment of all 8, 16, or 32-byte data onto 8-byte boundaries. Option `-dbl_align_all` causes all data to be aligned on 8-byte boundaries. Programs that depend on the use of these options may not be portable.

4.3 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.

4.3.1 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`.

4.3.2 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.

4.3.3 Declaring 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.

4.3.4 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.

4.3.5 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.

4.3.6 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.

4.3.7 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

4.4 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.

4.5 Unsigned Integers

The Fortran 95 compiler accepts a new data type, UNSIGNED, as an extension to the language. Four KIND parameter values are accepted with UNSIGNED: 1, 2, 4, and 8, corresponding to 1-, 2-, 4-, and 8-byte unsigned integers, respectively.

The form of an unsigned integer constant is a digit-string followed by the upper or lower case letter U, optionally followed by an underscore and kind parameter. The following examples show the maximum values for unsigned integer constants:

```
255u_1
65535u_2
4294967295U_4
18446744073709551615U_8
```

Expressed without a kind parameter (12345U), the default is the same as for default integer. This is U_4 but can be changed by the -xtypemap option, which will change the kind type for default unsigned integers.

Declare an unsigned integer variable or array with the UNSIGNED type specifier:

```
UNSIGNED U
UNSIGNED(KIND=2) :: A
UNSIGNED*8 :: B
```

4.5.1 Arithmetic Expressions

- Binary operations, such as + - * / cannot mix signed and unsigned operands. That is, `U*N` is illegal if `U` is declared `UNSIGNED`, and `N` is a signed `INTEGER`.
  - Use the `UNSIGNED` intrinsic function to combine mixed operands in a binary operation, as in `U*UNSIGNED(N)`
  - An exception is when one operand is an unsigned integer and the other is a signed integer constant expression with positive or zero value; the result is an unsigned integer.
  - The kind of the result of such a mixed expression is the largest kind of the operands.
- Exponentiation of a signed value is signed while exponentiation of an unsigned value is unsigned.
- Unary minus of an unsigned value is unsigned.
- Unsigned operands may mix freely with real, complex operands. (Unsigned operands cannot be mixed with interval operands.)

4.5.2 Relational Expressions

Signed and unsigned integer operands may be compared using intrinsic relational operations. The result is based on the unaltered value of the operands.

4.5.3 Control Constructs

- The CASE construct accepts unsigned integers as case-expressions.
- Unsigned integers are not permitted as DO loop control variables, or in arithmetic IF control expressions.

4.5.4 Input/Output Constructs

- Unsigned integers can be read and written using the I, B, O, and Z edit descriptors.
- They can also be read and written using list-directed and namelist I/O. The written form of an unsigned integer under list-directed or namelist I/O is the same as is used for positive signed integers.
- Unsigned integers can also be read or written using unformatted I/O.

4.5.5 Intrinsic Functions

- Unsigned integers are allowed in intrinsic functions, except for `SIGN` and `ABS`.
- A new intrinsic function, `UNSIGNED`, is analogous to `INT` but produces a result of unsigned type. The form is

  `UNSIGNED(`*v* [`,`*kind*] `)`.
- Another new intrinsic function, `SELECTED_UNSIGNED_KIND(` *var*`)`, returns the kind parameter for *var*.
- The `MIN` and `MAX` functions do not allow both signed and unsigned integer operands unless there is at least one operand of `REAL` type.
- Unsigned arrays cannot appear as arguments to array intrinsic functions.

4.6 Fortran 2003 Features

A number of features proposed in the Fortran 2003 draft standard appear in this release of the `f95` compiler.

4.6.1 Interoperability with C Functions

The new draft standard for Fortran provides:

- a means of referencing C language procedures and, conversely, a means of specifying that a Fortran subprogram can be referenced from a C function, and
- a means of declaring global variables that are linked with external C variables

The `ISO_C_BINDING` module provides access to named constants that are kind type parameters representing data that is compatible with C types.

The draft standard also introduces the `BIND(C)` attribute. A Fortran derived type is interoperable with C if it has the `BIND` attribute.

This release of the Fortran 95 compiler implements these features as described in the chapter 15 of the draft standard. Fortran also provides facilities for defining derived types, enumerations, and type aliases that correspond to C types, as described in chapter 4 of the draft standard.

4.6.2 IEEE Floating-Point Exception Handling

New intrinsic modules `IEEE_ARITHMETIC`, and `IEEE_FEATURES` provide support for exceptions and IEEE arithmetic in the Fortran language. Full support of these features is provided by:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
USE, INTRINSIC :: IEEE_FEATURES
```

These modules define a set of derived types, constants, rounding modes, inquiry functions, elemental functions, kind functions, and elemental and non-elemental subroutines. The details are contained in Chapter 14 of the draft standard for Fortran 2000.

4.6.3 Command-Line Argument Intrinsics

The Fortran 2003 draft standard introduces three new intrinsics for processing command-line arguments and environment variables. These are:

- `GET_COMMAND(`*command, length, status*`)`

  Returns in *command* the entire command line that invoked the program.

- `GET_COMMAND_ARGUMENT(`*number, value, length, status*`)`

  Returns a command-line argument in *value*.

- `GET_ENVIRONMENT_VARIABLE(`*name, value, length, status, trim_name*`)`

  Return the value of an environment variable.

4.6.4 `PROTECTED` Attribute

The Fortran 95 compiler now accepts the Fortran 2003 `PROTECTED` attribute. `PROTECTED` imposes limitations on the usage of module entities. Objects with the `PROTECTED` attribute are only definable within the module that declares them.

4.6.5 Fortran 2003 Asynchronous I/O

The compiler recognizes the `ASYNCHRONOUS` specifier on I/O statements:

```
ASYNCHRONOUS=['YES' | 'NO']
```

This syntax is as proposed in the Fortran 2000 draft standard, Chapter 9. In combination with the WAIT statement it allows the programmer to specify I/O processes that may be overlapped with computation. While the compiler recognizes ASYNCHRONOUS='YES', the draft standard does not require actual asynchronous I/O. In this release of the compiler, I/O is always synchronous.

4.6.6 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.

4.6.7 VALUE Attribute

The f95 compiler recognizes the VALUE type declaration attribute. This attribute has been proposed for the Fortran 2003 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
```

4.6.8 Fortran 2003 Stream I/O

A new "stream" I/O scheme has been proposed as part of the Fortran 2003 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*.

4.6.9 Fortran 2003 Formatted I/O Features

Three new Fortran 2003 formatted I/O specifiers have been implemented in `f95`. They may appear on `OPEN`, `READ`, `WRITE`, `PRINT`, and `INQUIRE` statements:

- `DECIMAL=['POINT'|'COMMA']`

  Change the default decimal editing mode. The default uses a period to separate the whole number and decimal parts of a floating-point number formatted with `D`, `E`, `EN`, `ES`, `F`, and `G` editing. `'COMMA'` changes the default to use comma instead of a period, to print, for example, `123,456`. The default is `'POINT'`, which uses a period, to print, for example, `123.456`.

- `ROUND=['PROCESSOR_DEFINED' | 'COMPATIBLE']`

  Set the default rounding mode for formatted I/O `D`, `E`, `EN`, `ES`, `F`, and `G` editing. With `'COMPATIBLE'`, the value resulting from data conversion is the one closer to the two nearest represetnations, or the value away from zero if the value is halfway between them. With `'PROCESSOR_DEFINED'`, the rounding mode is dependent on the processor's default mode, and is the compiler default if `ROUND` is not specified.

  As an example, `WRITE(*,'(f11.4)') 0.11115` prints `0.1111` in default mode, and `0.1112` in `'COMPATIBLE'` mode.

- `IOMSG=`*character-variable*

  Returns an error message as a string in the specified character variable. This is the same error message that would appear on standard output. Users should allocated a character buffer large enough to hold the longest message. (`CHARACTER*256` should be sufficient.)

When used in `INQUIRE` statements, these specifiers declare a character variable for returning the current values.

New edit descriptors `DP`, `DC`, `RP`, and `RC` change the defaults within a single `FORMAT` statement to decimal point, decimal comma, processor-defined rounding, and compatible rounding respectively. For example:

```
WRITE(*,'(I5,DC,F10.3)') N, W
```

prints a comma instead of a period in the `F10.3` output item.

See also the `-iorounding` compiler command-line option for changing the floating-point rounding modes on formatted I/O. (Section , "`-iorounding[={compatible|processor-defined}]`" on page 3-32.)

4.7 Additional I/O Extensions

The section describes extensions to Fortran 95 Input/Output handling that are accepted by the `f95` compiler that are not part of the Fortran 2003 draft standard. Some are I/O extensions that appeared in the Fortran 77 compiler, `f77`, and are now part of the Fortran 95 compiler.

4.7.1 I/O Error Handling Routines

Two new functions enable the user to specify their own error handling routine for formatted input on a logical unit. When a formatting error is detected, the runtime I/O library calls the specified user-supplied handler routine with data pointing at the character in the input line causing the error. The handler routine can supply a new character and allow the I/O operation to continue at the point where the error was detected using the new character; or take the default Fortran error handling.

The new routines, `SET_IO_ERR_HANDLER(3f)` and `GET_IO_ERR_HANDLER(3f)`, are module subroutines and require `USE SUN_IO_HANDLERS` in the routine that calls them. See the man pages for these routines for details.

4.7.2 Variable Format Expressions

Fortran 77 allowed any integer constant in a format to be replaced by an arbitrary expression enclosed in angle brackets:

```
1  FORMAT( ... < expr > ... )
```

Variable format expressions cannot appear as the *n* in an *n*H... edit descriptor, in a `FORMAT` statement that is referenced by an `ASSIGN` statement, or in a `FORMAT` statement within a parallel region.

This feature is enabled natively in `f95`, and does not require the `-f77` compatibility option flag.

4.7.3 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.

4.7.4 Binary Unformatted I/O

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.

4.7.5 Miscellaneous I/O Extensions

- 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*.
- Non-advancing I/O is enabled with ADVANCE='NO', as in:

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

4.8 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 parallelization directives are the default with `f95 -explicitpar`. To switch to Cray-style directives, use the `-mp=cray` compiler command-line flag. Explicit parallelization with OpenMP directives requires compiling with `-openmp`.

A complete summary of Fortran directives appears in Appendix D.

4.8.1 Form of Special `f95` Directive Lines

`f95` recognizes its own special directives in addition to those described in Chapter 2. These have the following syntax:

```
!DIR$ d1, d2, …
```

4.8.1.1 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.

4.8.1.2 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.

4.8.2 `FIXED` and `FREE` Directives

These directives specify the source form of lines following the directive line.

4.8.2.1 Scope

They apply to the rest of the *file* in which they appear, or until the next `FREE` or `FIXED` directive is encountered.

4.8.2.2 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.

4.8.2.3 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
```

4.8.3 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 D and described in the chapter on parallelization in the *Fortran Programming Guide*. f95 recognizes both Sun and Cray style parallelization directives, as well as the OpenMP Fortran API directives. OpenMP parallelization is described in the *OpenMP API User's Guide*.

4.9 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 directory specified by the -moddir=*dir* flag or the MODDIR evironment variable. If not specified, the compiler writes .mod files in the current working directory.

The compiler 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.

In this example, all the files are compiled at once. The module source files appear first before their use in the main program.

```
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
```

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.

4.9.1 Searching for Modules

With the release of the Sun ONE Studio 7 Fortran 95 compiler, `.mod` files can be stored into an archive (`.a`) file. An archive must be explicitly specified in a `-M`*path* flag on the command line for it to be searched for modules. The compiler does not search archive files by default.

Only `.mod` files with the same names that appear on `USE` statements will be searched. For example, the Fortran 95 statement `USE mymod` causes the compiler to search for the module file `mymod.mod` by default.

While searching, the compiler gives higher priority to the directory where the module files are being written. This can be controlled by the `-moddir=`*dir* option flag and the `MODDIR` environment variable. This implies that if only the `-M`*path* option is specified, the current directory will be searched for modules first, before the directories and files listed on the `-M` flag.

4.9.2 The `-use=`*list* Option Flag

The `-use=`*list* flag forces one or more implicit `USE` statements into each subprogram or module subprogram compiled with this flag. By using the flag, it is not necessary to modify source programs when a module or module file is required for some feature of a library or application.

Compiling with `-use=`*module_name* has the effect of adding a `USE` *module_name* to each subprogram or module being compiled. Compiling with `-use=`*module_file_name* has the effect of adding a `USE` *module_name* for each of the modules contained in the *module_file_name* file.

4.9.3 The `fdumpmod` Command

Use the `fdumpmod`(1) command to display information about the contents of a compiled module information file.

```
demo% fdumpmod x.mod group.mod
x 1.0 v8,i4,r4,d8,n16,a4 x.mod
group 1.0 v8,i4,r4,d8,n16,a4 group.mod
```

The `fdumpmod` command will display information about modules in a single `.mod` file, files formed by concatenating `.mod` files, and in `.a` archives of `.mod` files. The display includes the name of the module, a version number, the target architecture, and flags indicating compilation options with which the module is compatible. See the `fdumpmod`(1) man page for details.

4.10 Intrinsics

`f95` supports some intrinsic procedures that are extensions beyond the standard.

TABLE 4-4 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.

See the *Fortran Library Reference* for a more complete discussion of intrinsics, including those from Fortran 77 that are recognized by the Fortran 95 compiler.

4.11 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.

4.12 Mixing Languages

On Solaris systems, routines written in C can be combined with Fortran programs, since these languages have common calling conventions.See the C-Fortran Interface chapter in the *Fortran Programming Guide* for details on how to interoperate between C and Fortran routines.

CHAPTER 5

FORTRAN 77 Compatibility: Migrating to Fortran 95

The Fortran 95 compiler, `f95`, will compile most legacy FORTRAN 77 programs, including programs utilizing non-standard extensions previously compiled by the `f77` compiler.

`f95` will accept many of these FORTRAN 77 features directly. Others require compiling in FORTRAN 77 compatibility mode (`f95 -f77`).

This chapter describes the FORTRAN 77 features accepted by `f95`, and lists those `f77` features that are incompatible with `f95`. For details on any of the non-standard FORTRAN 77 extensions that were accepted by the `f77` compiler, see the *FORTRAN 77 Language Reference* manual at `http://docs.sun.com/source/806-3594/index.html`.

See Chapter 4 for other extensions to the Fortran 95 language accepted by the `f95` compiler.

`f95` will compile standard-conforming FORTRAN 77 programs. To ensure continued portability, programs utilizing non-standard FORTRAN 77 features should migrate to standard-conforming Fortran 95. Compiling with the `-ansi` option will flag all non-standard usages in your program.

5.1 Compatible `f77` Features

`f95` accepts the following non-standard features of the FORTRAN 77 compiler, `f77`, either directly or when compiling in `-f77` compatibility mode:

- **Source Format**
  - Continuation lines can start with '`&`' in column 1. [`-f77=misc`]
  - The first line in an include file can be a continuation line. [`-f77=misc`]
  - Use `f77` tab-format. [`-f77=tab`]

- Tab-formatting can extend source lines beyond column 72. [`-f77=tab`]
- `f95` tab-formatting will not pad character strings to column 72 if they extend over a continuation line. [`-f77`]

- **I/O:**
  - You can open a file with `ACCESS='APPEND'` in Fortran 95.
  - List-directed output uses formats similar to the `f77` compiler. [`-f77=output`]
  - `f95` allows `BACKSPACE` on a direct-access file, but not `ENDFILE`.
  - `f95` allows implicit field-width specifications in format edit descriptors. For example, `FORMAT(I)` is allowed.
  - `f95` will recognize `f77` escape sequences (for example, `\n \t \'`) in output formats. [`-f77=backslash`.]
  - `f95` recognizes `FILEOPT=` in `OPEN` statements.
  - `f95` allows `SCRATCH` files to be closed with `STATUS='KEEP'` [`-f77`]. When the program exits, the scratch file is not deleted. `SCRATCH` files can also be opened with `FILE=`*name* when compiled with `-f77`.
  - Direct I/O is permitted on internal files. [`-f77`]
  - `f95` recognizes FORTRAN 77 format edit descriptors `A`, `$`, and `SU`. [`-f77`]
  - `FORM='PRINT'` can appear on `OPEN` statements. [`-f77`]
  - `f95` recognizes the legacy FORTRAN input/output statements `ACCEPT` and `TYPE`.
  - Compile with `-f77=output` to write FORTRAN 77 style `NAMELIST` output.
  - A `READ` with only `ERR=` specified (no `IOSTAT=` or `END=` branches) treats the `ERR=` branch as an `END=` when an EOF is detected. [`-f77`]
  - VMS Fortran `NAME='`*filename*`'` is accepted on `OPEN` statements. [`-f77`]
  - `f95` accepts an extra comma after `READ()` or `WRITE()`. [`-f77`]
  - `END=` branch can appear on direct access `READ` with `REC=`. [`-f77=input`]
  - Allow format edit descriptor `E`*w*`.`*d*`.`*e* and treat it as `E`*w*`.`*d*`.E`*e*. [`-f77`]
  - Character strings can be used in the FORMAT of an input statement. [`-f77=input`]
  - `IOSTAT=` specifier can appear in `ENCODE/DECODE` statements.
  - List-directed I/O is allowed with `ENCODE/DECODE` statements.
  - Asterisk (`*`) can be used to stand in for `STDIN` and `STDOUT` when used as a logical unit in an I/O statement.
  - Arrays can appear in the `FMT=` specifier. [`-f77=misc`]
  - `PRINT` statement accepts namelist group names. [`-f77=output`]
  - The compiler accepts redundant commas in `FORMAT` statements.

- While performing NAMELIST input, entering a question mark (?) responds with the name of the namelist group being read. [-f77=input]

- **Data Types, Declarations, and Usage:**
  - In a program unit, the IMPLICIT statement may follow any other declarative statement in the unit.
  - f95 accepts the IMPLICIT UNDEFINED statement.
  - f95 accepts the AUTOMATIC statement, a FORTRAN 77 extension.
  - f95 accepts the STATIC statement and treats it like a SAVE statement.
  - f95 accepts VAX STRUCTURE, UNION, and MAP statements.(See Section 4.4, "STRUCTURE and UNION (VAX Fortran)" on page 4-12)
  - LOGICAL and INTEGER variables can be used interchangeably. [-f77=logical]
  - INTEGER variables can appear in conditional expressions, such as DO WHILE. [-f77=logical]
  - Cray pointers can appear in calls to intrinsic functions.
  - f95 will accept data initializations using slashes on type declarations. For example: REAL MHW/100.101/, ICOMX/32.223/
  - f95 allows assigning Cray character pointers to non-pointer variables and to other Cray pointers that are not character pointers.
  - f95 allows the same Cray pointer to point to items of different type sizes (for example, REAL*8 and INTEGER*4).
  - A Cray pointer can be declared INTEGER in the same program unit where it is declared as a POINTER. The INTEGER declaration is ignored. [-f77=misc]
  - A Cray pointer can be used in division and multiplication operations. [-f77=misc]
  - Variables in an ASSIGN statement can be of type INTEGER*2. [-f77=misc]
  - Expressions in alternate RETURN statements can be non-integer type. [-f77=misc]
  - Variables with the SAVE attribute can be equivalenced to an element of a COMMON block.
  - Initializers for the same array can be of different types. Example: REAL*8 ARR(5) /12.3 1, 3, 5.D0, 9/
  - Type declarations for namelist items can follow the NAMELIST statement.
  - f95 accepts the BYTE data type.
  - f95 allows non-integers to be used as array subscripts. [-f77=subscript]
  - f95 allows relational operators .EQ. and .NE. to be used with logical operands. [-f77=logical]
  - f95 will accept the legacy f77 VIRTUAL statement, and treats it as a DIMENSION statement.

- Different data structures can be equivalenced in a manner that is compatible with the `f77` compiler. [`-f77=misc`]
- Like the `f77` compiler, `f95` allows many intrinsics to appear in initialization expressions on `PARAMETER` statements.
- `f95` allows assignment of an integer value to `CHARACTER*1` variables. [`-f77=misc`]
- BOZ constants can be used as exponents. [`-f77=misc`]
- BOZ constants can be assigned to character variables. For example:
```
character*8 ch
ch ="12345678"X
```
- BOZ constants can be used as arguments to intrinsic function calls. [`-f77=misc`]
- A character variable can be initialized with an integer value in a `DATA` statement. The first character in the variable is set to the integer value and the rest of the string, if longer than one character, is blank-filled.
- An integer array of hollerith characters can be used as a format descriptor. [`-f77`].
- Constant folding will not be done at compile time if it results in a floating-point exception. [`-f77=misc`]
- When compiling with `-f77=misc`, `f95` will automatically promote a `REAL` constant to the appropriate kind (`REAL*8` or `REAL*16`) in assignments, data, and parameter statements, in the manner of the `f77` compiler. [`-f77=misc`]
- Equivalenced variables are allowed on an assigned GOTO. [`-f77`]
- Non-constant character expressions can be assigned to numeric variables.
- Compiling with `-f77=misc` allows `*`*kind* after the variable name in type declarations. [`-f77=misc`]. For example
```
REAL Y*4, X*8(21))
INTEGER FUNCTION FOO*8(J)
```
- A character substring may appear as an implied-DO target in a `DATA` statement. [`-f77=misc`]
  For example: `DATA (a(i:i), i=1,n) /n*'+'/`
- Integer expressions within parentheses can appear as a type size. For example:
```
PARAMETER (N=2)
INTEGER*(N+2) K
```

- **Programs, Subroutines, Functions, and Executable Statements:**
  - `f95` does not require a `PROGRAM` statement to have a *name*.
  - Functions can be called by a `CALL` statement as if they were subroutines. [`-f77`]
  - Functions do not have to have their return value defined. [`-f77`]

- An alternate return specifier (**label* or *&label*) can appear in the actual parameter list and in different positions. [`-f77=misc`]
- `%VAL` can be used with an argument of type `COMPLEX`. [`-f77=misc`]
- `%REF` and `%LOC` are available. [`-f77=misc`]
- A subroutine can call itself recursively without declaring itself with a `RECURSIVE` keyword. [`-f77=misc`] However, programs that perform indirect recursion (routine A calls routine B which then calls routine A) should also be compiled with the `-xrecursive` flag to work correctly.
- A subroutine with alternate returns can be called even when the dummy argument list does not have an alternate return list.
- Compiling with `-f77=misc` allows statement functions to be defined with arguments typed other than `INTEGER` or `REAL`, and actual arguments will be converted to the type defined by the statement function. [`-f77=misc`]
- Allow null actual arguments. For example: `CALL FOO(I,,,J)` has two null arguments between the first `I` and the final `J` argument.
- `f95` treats a call to the function `%LOC()` as a call to `LOC()`. [`-f77=misc`]
- Allow unary plus and unary minus after another operator such as `**` or `*`.
- Allow a second argument with the `CMPLX()` intrinsic function even when the first argument is a `COMPLEX` type. In this case, the real part of the first argument is used. [`-f77=misc`]
- Allow the argument to the `CHAR()` intrinsic function to exceed 255 with only a warning, not an error. [`-f77=misc`]
- Allow negative shift counts with only a warning, not an error.
- Search for an `INCLUDE` file in the current directory as well as those specified in the `-I` option. [`-f77=misc`]
- Allow consecutive `.NOT.` operations, such as `.NOT..NOT..NOT.(I.EQ.J).` [`-f77=misc`]

- **Miscellaneous**
  - The `f95` normally does not issue progress messages to standard out. The `f77` compiler did issue progress messages, displaying the names of the routines it was compiling. This convention is retained when compiling with the `-f77` compatibility flag.
  - Programs compiled by the `f77` compiler did not trap on arithmetic exceptions, and automatically called `ieee_retrospective` on exit to report on any exceptions that may have occured during execution. Compiling with the `-f77` flag mimics this behavior of the `f77` compiler. By default, the `f95` compiler traps on the first arithmetic exception encountered and does not call `ieee_retrospective`.

- The `f77` compiler treated a `REAL*4` constant as if it had higher precision in contexts where higher precision was needed. When compiling with the `-f77` flag, the `f95` compiler allows a `REAL*4` constant to have double or quad precision when the constant is used with a double or quad precision operand, respectively.
- Allow the DO loop variable to be redefined within the loop. [`-f77=misc`]
- Display the names of program units being compiled. [`-f77=misc`]
- Allow the types of the variables used in a DIMENSION statement to be declared after the DIMENSION statement. Example:

```
SUBROUTINE FOO(ARR,G)
DIMENSION ARR(G)
INTEGER G
RETURN
END
```

For details on the syntax and semantics of non-standard language extensions, see the *FORTRAN 77 Language Reference* at http://docs.sun.com/source/806-3594/index.html.

5.2 Incompatibility Issues

The following lists known incompatibilities that arise when compiling and testing legacy `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`.

- **Source Format**
  - An ANSI warning is given for names longer than 6 characters when the `-f77` option is specified.
- **I/O:**
  - `f95` does not allow `ENDFILE` on a direct-access file.
  - `f95` does not recognize the `'`*n* form for specifying a record number in direct access I/O: `READ (2 '13) X,Y,Z`
  - `f95` does not recognize the legacy `f77` "`R`" format edit descriptor.
  - `f95` does not allow the `DISP=` specifier in a `CLOSE` statement.
  - Bit constants are not allowed on a `WRITE` statement.
  - Fortran 95 `NAMELIST` does not allow arrays and character strings with variable lengths.
  - Opening a direct access file with `RECL=1` cannot be used as a "stream" file. Use `FORMAT='STREAM'` instead.

- Fortran 95 reports illegal I/O specifiers as errors. `f77` gave only warnings.

- **Data Types, Declarations, and Usage:**
  - `f95` allows only 7 array subscripts; `f77` allowed 20.
  - `f95` does not allow non-constants in `PARAMETER` statements.
  - Integer values cannot be used in the initializer of a `CHARACTER` type declaration.
  - The `REAL()` intrinsic returns the real part of a complex argument instead of converting the argument to `REAL*4`. This gives different results when the argument is `DOUBLE COMPLEX` or `COMPLEX*32`
  - Fortran 95 will not allow array elements in boundary expressions before the array is declared. For example:

```
subroutine s(i1,i2)
integer i1(i2(1):10)
dimension i2(10)
...ERROR: "I2" has been used as a function, therefore it must
not be declared with the explicit-shape DIMENSION attribute.

end
```

- **Programs, Subroutines, Functions, Statements:**
  - The maximum length for names is 127 characters.
- **Command-line Options:**
  - `f95` does not recognize the `f77` compiler options `-dbl -oldstruct -i2 -i4` and some suboptions of `-vax`.
- `f77` **Library Routines Not Supported by** `f95`**:**
  - The POSIX library.
  - The `IOINIT()` library routine.
  - The tape I/O routines `topen, tclose, twrite, tread, trewin, tskipf, tstate`.
  - `start_iostats` and `end_iostats` library routines.
  - `f77_init()` function.
  - `f95` does not allow the `IEEE_RETROSPECTIVE` subroutine to be bypassed by defining the user's own routine with the same name.

5.3 Linking With f77-Compiled Routines

- To mix f77 and f95 object binaries, link with f95 compile with the -xlang=f77 option. Perform the link step with f95 even if the main program is an f77 program
- Example: Compiling an f95 main program with an f77 object file.

```
demo% cat m.f95
CHARACTER*74 :: c = 'This is a test.'
   CALL echo1( c )
END
demo% f95 -xlang=f77 m.f95 sub77.o
demo% a.out
 This is a test.
demo%
```

- The FORTRAN 77 library and intrinsics are available to f95 programs and are listed in the *Fortran Library Reference Manual*.

  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).

5.3.1 Fortran 95 Intrinsics

The Fortran 95 standard supports intrinsic functions that FORTRAN 77 did not have. The full set of Fortran 95 intrinsics, including non-standard intrinsics, appears in the *Fortran Library Reference* manual.

If you use any of the intrinsic names listed in the *Fortran Library Reference* as a function name in your program, you must add an `EXTERNAL` statement for `f95` to use your routine rather than the intrinsic one.

The *Fortran Library Reference* also lists all the intrinsics recognized by earlier releases of the `f77` compiler. The `f95` compiler recognizes these names as intrinsics as well.

Compiling with `-f77=intrinsics` limits the compiler's recognition of intrinsic functions to just those that were known to the `f77` compiler, ignoring the Fortran 95 intrinsics.

5.4 Additional Notes About Migrating to the `f95` Compiler

- The `floatingpoint.h` header file replaces `f77_floatingpoint.h`, and should be used in source programs as follows:

  ```
  #include "floatingpoint.h"
  ```

- Header file references of the form `f77/`*filename* should be changed to remove the `f77/` directory path.
- Some programs utilizing non-standard aliasing techniques (by overindexing arrays, or by overlapping Cray or Fortran pointers) may benefit by compiling with the appropriate `-xalias` flag. See Section , "`-xalias`[=keywords]" on page 3-54. This is discussed with examples in the chapter on porting "dusty deck" programs in the *Fortran Programming Guide*.

APPENDIX **A**

Runtime Error Messages

This appendix describes the error messages generated by the Fortran 95 runtime I/O library and operating system.

A.1 Operating System Error Messages

Operating system error messages include system call failures, C library errors, and shell diagnostics. The system call error messages are found in `intro`(2). System calls made through the `Fortran` library do not produce error messages directly. The following system routine in the `Fortran` library calls C library routines which produce an error message:

```
    integer system, status
    status = system("cp  afile bfile")
    print*, "status = ", status
    end
```

The following message is displayed:

```
cp: cannot access afile
 status =  512
```

A.2 f95 Runtime I/O Error Messages

The f95 I/O library issues diagnostic messages when errors are detected at runtime. Here is an 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-1 lists the runtime I/O messages issued by f95.

TABLE A-1 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
1010	record too long

TABLE A-1 `f95` Runtime I/O Messages *(Continued)*

Error	Message
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
1057	empty internal file
1058	list-directed I/O on unformatted unit

TABLE A-1 `f95` Runtime I/O Messages *(Continued)*

Error	Message
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
1086	unexpected character in logical value
1087	unexpected character in character value
1088	unexpected character before NAMELIST group name

TABLE A-1 f95 Runtime I/O Messages *(Continued)*

Error	Message
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
1125	unassociated READWRITE specifier
1126	unassociated SEQUENTIAL specifier
1127	unassociated STATUS specifier
1128	unassociated UNFORMATTED specifier

TABLE A-1 f95 Runtime I/O Messages *(Continued)*

Error	Message
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
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

TABLE A-1 f95 Runtime I/O Messages *(Continued)*

Error	Message
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
1191	unallocated array passed to array intrinsic function
1192	illegal rank
1193	small source size
1194	zero array size
1195	negative elements in shape

TABLE A-1 `f95` Runtime I/O Messages *(Continued)*

Error	Message
1196	`illegal kind`
1197	`nonconformable array`
1213	`asynchronous I/O on unconnected unit`
1214	`asynchronous I/O on synchronous unit`
1215	`a data edit descriptor and I/O list item type are incompatible`
1216	`current I/O list item doesn't match with any data edit descriptor`
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`
2021	`lock not initialized` (OpenMP)
2122	`deadlock in using lock variable` (OpenMP)
2123	`lock not set` (OpenMP)

APPENDIX **B**

Features Release History

This Appendix lists the new and changed features in this release and previous releases of the Fortran 95 compiler.

The Fortran 95 compiler, version 8.1, is a component released with Sun Studio 10.

B.1 Sun Studio 10 Fortran Release:

- **Compiling for AMD-64 Processors**

  This release introduces `-xarch=amd64` and `-xtarget=opteron` for compiling applications to run on 64-bit x86 platforms.

- **File sharing between big-endian and little-endian platforms**

  The new compiler flag `-xfilebyteorder` provides cross-platform support of binary I/O files.

- **OpenMP available on Solaris OS x86 platforms**

  With this release of Sun Studio, the OpenMP API for shared-memory parallelism is available on Solaris x86 platforms as well as Solaris SPARC platforms. The same functionality is now enabled on both platforms.

- **OpenMP option `-openmp=stubs` no longer supported**

  An OpenMP "stubs" library is provided for user's convenience. To compile an OpenMP program that calls OpenMP library functions but ignores the OpenMP pragmas, compile the program with the -openmp option and link the object files with the libompstubs.a library. For example:

  ```
  % f95 omp_ignore.c -lompstubs
  ```

  Linking with both `libompstubs.a` and the OpenMP runtime library `libmtsk.so` is unsupported and may result in unexpected behavior.

B.2 Sun Studio 9 Fortran Release:

- **Fortran 95 compiler released on x86 Solaris platforms:**

  This release of Sun Studio makes the Fortran 95 compiler available on Solaris OS x86 platforms. Compile with `-xtarget` values `generic`, `native`, `386`, `486`, `pentium`, `pentium_pro`, `pentium3`, or `pentium4`, to generate executables on Solaris x86 platforms. The default on x86 platforms is `-xtarget=generic`.

  The following `f95` features are not yet implemented on x86 platforms and are only available on SPARC platforms:

  - Interval Arithmetic (compiler options `-xia` and `-xinterval`)
  - Quad (128-bit) Arithmetic (for example, REAL*16)
  - IEEE Intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES
  - The `sun_io_handler` module
  - Parallelization options such as `-autopar`, `-parallel`, `-explitipar`, and `-openmp`.

  The following `f95` command-line options are only available on x86 platforms and not on SPARC platforms: `-fprecision`, `-fstore`, `-nofstore`

  The following `f95` command-line options are only available on SPARC platforms and not on x86 platforms: `-xcode`, `-xmemalign`, `-xprefetch`, `-xcheck`, `-xia`, `-xinterval`, `-xipo`, `-xjobs`, `-xlang`, `-xlinkopt`, `-xloopinfo`, `-xpagesize`, `-xprofile_ircache`, `-xreduction`, `-xvector`, `-depend`, `-openmp`, `-parallel`, `-autopar`, `-explicitpar`, `-vpara`, `-XlistMP`.
  Also, on x86 platforms `-fast` adds `-nofstore`.

- **Improved Runtime Performance:**

  Runtime performance for most applications should improve significantly with this release. For best results, compile with high optimization levels `-xO4` or `-xO5`. At these levels the compiler may now inline contained procedures, and those with assumed-shape, allocatable, or pointer arguments.

- **Fortran 2003 Command-Line Intrinsics:**

  The Fortran 2003 draft standard introduces three new intrinsics for processing command-line arguments and environment variables. These have been implemented in this release of the `f95` compiler. The new intrinsics are:

  - `GET_COMMAND(`*command, length, status*`)`

    Returns in command the entire command line that invoked the program.

  - `GET_COMMAND_ARGUMENT(`*number, value, length, status*`)`

    Returns a command-line argument in value.

- GET_ENVIRONMENT_VARIABLE(*name, value, length, status, trim_name*)

  Return the value of an environment variable.

- **New and Changed Command-Line Options:**

  The following `f95` command-line options are new in this release. See the Chapter 3 for details.

  - `-xipo_archive={ none | readonly | writeback }`

    Allow crossfile optimization to include archive (`.a`) libraries. (SPARC only)

  - `-xprefetch_auto_type=[no%]indirect_array_access`

    Generate indirect prefetches for a data arrays accessed indirectly. (SPARC only)

  - `-xprofile_pathmap=`*collect_prefix*`:`*use_prefix*

    Set path mapping for profile data files. Use the `-xprofile_pathmap` option with the `-xprofile=use` option when profiling into a directory that is not the directory used when previously compiling with `-xprofile=collect`.

  The following command-line option defaults have changed with this release of `f95`.

  - The default for `-xprefetch` is `-xprefetch=no%auto,explicit`.
  - The default for `-xmemalign` is `-xmemalign=8i`; when compiling with one of the `-xarch=v9` options the default is `-xmemalign=8f`.
  - The default for -xcode when compiling with one of the `-xarch=v9` options is `abs44`.

    To compile with the defaults used in previous compiler releases, specify the following options explicitly:

    `-xarch=v8 -xmemalign=4s -xprefetch=no` for 32-bit compilation
    `-xcode=abs64 -xprefetch=no` for 64-bit compilation

- **Default SPARC Architecture is V8PLUS:**

  The default SPARC architecture is no longer V7. Support for `-xarch=v7` is limited in this Sun Studio 9 release. The new default is V8PLUS (UltraSPARC). Compiling with `-xarch=v7` is treated as `-xarch=v8` because the Solaris 8 OS only supports `-xarch=v8` or better.

  To deploy on SPARC V8 systems (for example, SPARCStation 10), compile with `-xarch=v8` explicitly. The provided system libraries run on SPARC V8 architectures.

  To deploy on SPARC V7 systems (for example, SPARCStation 1), compile with `-xarch=v7` explicitly. The provided system libraries use the SPARC V8 instruction set. For the Sun Studio 9 release, only the Solaris 8 OS supports the SPARC V7 architecture. When a SPARC V8 instruction is encountered, the OS interprets the instruction in software. The program will run, but performance will be degraded.

- **OpenMP: Maximum Number of Threads Increased:**

The maximum number of threads for `OMP_NUM_THREADS` and the multitasking library has increased from 128 to 256.

- **OpenMP: Automatic Scoping of Variables:**

  This release of the Fortran 95 compiler's implementation of the OpenMP API for shared-memory parallel programming features automatic scoping of variables in parallel regions. See the OpenMP API User's Guide for details. (OpenMP is only implemented on SPARC platforms for this release.)

B.3 Sun Studio 8 Fortran Release:

- **Enhanced `-openmp` option:**

  The `-openmp` option flag has been enhanced to facilitate debugging OpenMP programs. To use `dbx` to debug your OpenMP application, compile with

  ```
  -openmp=noopt -g
  ```

  You will then be able to use `dbx` to breakpoint within parallel regions and display contents of variables. See Section , "`-openmp[={parallel|noopt|none}]`" on page 3-41.

- **Multi-process compilation:**

  Specify `-xjobs=`*n* with `-xipo` and the interprocedural optimizer will invoke at most *n* code generator instances to compile the files listed on the command line. This option can greatly reduce the build time of large applications on a multi-cpu machine. See Section , "`-xjobs=n`" on page 3-76.

- **Making assertions with `PRAGMA ASSUME`:**

  The `ASSUME` pragma is a new feature in this release of the compiler. This pragma gives hints to the compiler about conditions the programmer knows are true at some point in a procedure. This may help the compiler to do a better job optimizing the code. The programmer can also use the assertions to check the validity of the program during execution. See Section 2.3.1.9, "The `ASSUME` Directives" on page 2-13, and Section , "`-xassume_control[=keywords]`" on page 3-61.

- **More Fortran 2000 features:**

  The following features appearing in the Fortran 2000 draft standard have been implemented in this release of Fortran 95 compiler. These are described in Chapter 4.

  - Exceptions and IEEE Arithmetic:

    New intrinsic modules IEEE_ARITHMETIC, and IEEE_FEATURES provide support for exceptions and IEEE arithmetic in the Fortran language. See Section 4.6.2, "IEEE Floating-Point Exception Handling" on page 4-16.

- Interoperability with C:

  The new draft standard for Fortran provides a means of referencing C language procedures and, conversely, a means of specifying that a Fortran subprogram can be referenced from a C function. It also provides a means of declaring global variables that are linked with external C variables. See Section 4.6.1, "Interoperability with C Functions" on page 4-15.

- `PROTECTED` Attribute

  The Fortran 95 compiler now accepts the Fortran 2000 `PROTECTED` attribute. `PROTECTED` imposes limitations on the usage of module entities. Objects with the `PROTECTED` attribute are only definable within the module that declares them. Section 4.6.4, "`PROTECTED` Attribute" on page 4-16.

- `ASYNCHRONOUS` I/O Specifier

  The compiler recognizes the `ASYNCHRONOUS` specifier on I/O statements:

  ```
  ASYNCHRONOUS=['YES' | 'NO']
  ```

  This syntax is as proposed in the draft standard. See Section 4.6.5, "Fortran 2003 Asynchronous I/O" on page 4-16.

- **Enhanced compatibility with legacy `f77`:**

  A number of new features enhance the Fortran 95 compiler's compatibility with legacy Fortran 77 compiler, `f77`. These include variable format expressions (VFE's), long identifiers, `-arg=local` , and the `-vax` compiler option. See Chapter 3 and Chapter 4.

- **I/O error handlers:**

  Two new functions enable the user to specify their own error handling routine for formatted input on a logical unit. These routines are described in Section 4.7.1, "I/O Error Handling Routines" on page 4-19, and in man pages and the *Fortran Library Reference*.

- **Unsigned integers:**

  With this release, the Fortran 95 compiler accepts a new data type, `UNSIGNED`, as an extension to the language. See Section 4.5, "Unsigned Integers" on page 4-13.

- **Set preferred stack/heap page size:**

  A new command-line option, `-xpagesize`, enables the running program to set the preferred stack and heap page size at program startup. See Section , "`-xpagesize=size`" on page 3-83.

- **Faster and enhanced profiling:**

  This release introduces the new command-line option `-xprofile_ircache=`*path*, to speed up the "use" compilation phase during profile feedback. See Section , "`-xprofile_ircache[=path]`" on page 3-89. See also Section , "`-xprofile_pathmap=collect_prefix:use_prefix`" on page 3-90.

- **Enhanced "known libraries":**

The `-xknown_lib` option has been enhanced to include more routines from the Basic Linear Algebra library, BLAS. See Section , "-xknown_lib=library_list" on page 3-77.

- **Link-time Optimization:**

  Compile and link with the new `-xlinkopt` flag to invoke a post-optimizer to apply a number of advanced performance optimizations on the generated binary object code at link time. See Section , "-xlinkopt[={1|2|0}]" on page 3-79.

- **Initialization of local variables:**

  A new extension to the `-xcheck` option flag enables special initialization of local variables. Compiling with `-xcheck=init_local` initializes local variables to a value that is likely to cause an arithmetic exception if it is used before it is assigned by the program. See Section , "-xcheck=keyword" on page 3-63

B.4 Sun ONE Studio 7, Compiler Collection (Forte Developer 7) Release:

- **Fortran 77 Functionality Absorbed Into Fortran 95 Compiler**

  This release of the Forte Developer software replaces the `f77` compiler with added functionality in the `f95` compiler. The `f77` command is a script that calls the `f95` compiler:

> *the command:*
> `f77` *options files libraries*
> *becomes a call to the* `f95` *compiler::*
> `f95 -f77=%all -ftrap=%none` *options files* `-lf77compat` *libraries*

  See Chapter 5 for details on Fortran 77 compatibilities and incompatibilities.

- **Fortran 77 Compatibility Mode:**

  The new `-f77` flag selects various compatibility features that enable the compiler to accept many Fortran 77 constructs and conventions that are normally incompatible with Fortran 95. See Section , "`-f77`[=list]" on page 3-21, and Chapter 5.

- **Compiling "Dusty Deck" Programs That Employ Non-Standard Aliasing:**

  The `f95` compiler must assume that programs it compiles adhere to the Fortran 95 standard regarding aliasing of variables through subprogram calls, global variables, pointers, and overindexing. Many "dusty deck" legacy programs intentionally utilized aliasing techniques to get around shortcomings in early versions of the Fortran language. Use the new `-xalias` flag to advise the compiler about how far the program deviates from the standard and what kind of aliasing syndromes it should expect. In some cases the compiler generates correct code only when the proper `-xalias` suboption is specified. Programs that conform strictly to the standard will find some performance improvement by advising the compiler to be unconcerned about aliasing. See Section , "`-xalias`[=keywords]" on page 3-54, and the chapter on Porting in the *Fortran Programming Guide*.

- **Enhanced MODULE Features:**
  - New flag `-use=`*list* forces one or more implicit `USE` statements into each subprogram. See Section , "`-use=list`" on page 3-50.
  - New flag `-moddir=`*path* controls where the compiler writes compiled `MODULE` subprograms (`.mod` files). See Section , "`-moddir=path`" on page 3-36. A new environment variable, `MODDIR`, also controls where .mod files are written.

- The -M*path* flag will now accept directory paths, archive (`.a`) files, or module (`.mod`) files to search for `MODULE` subprograms. The compiler determines the type of the file by examining its contents; the actual file name extension is ignored. See Section , "-Mpath" on page 3-35.
- When searching for modules, the compiler now looks first in the directory where module files are being written.

See Section 4.9, "Module Files" on page 4-23 for details.

- **Enhanced Global Program Analysis With -Xlist:**

  This release of the `f95` compiler adds a number of new checks to the global program analysis provided by the `-Xlist` flag. The new `-XlistMP` suboption opens a new domain of static program analysis, verification of OpenMP parallelization directives. See Section , "-Xlist[x]" on page 3-52, the Forte Developer *OpenMP API User's Guide*, and the chapter on Program Analysis and Debugging in the *Fortran Programming Guide* for details.

- **Identifying Known Libraries With -xknown_lib=library:**

  A new option, `-xknown_lib=`*library*, directs the compiler to treat references to certain known libraries as intrinsics, ignoring any user-supplied versions. This enables the compiler to perform optimizations over library calls based on its special knowledge of the library. In this release, the known library names are limited to `blas`, for a subset of the BLAS routines in the Sun Performance Library, and `intrinsics`, for ignoring explicit `EXTERNAL` declarations for Fortran 95 standard intrinsics and any user-supplied versions of these routines. See Section , "-xknown_lib=library_list" on page 3-77.

- **Ignoring Dummy Argument Type in Interfaces:**

  A new directive, `!$PRAGMA IGNORE_TKR` {*list_of_variables*}, causes the compiler to ignore the type, kind, and rank for the specified dummy argument names appearing in a generic procedure interface when resolving a specific call. Using this directive greatly simplifies writing generic interfaces for wrappers that call specific library routines based on argument type, kind, and rank. See Section 2.3.1.2, "The IGNORE_TKR Directive" on page 2-10 for details.

- **Enhanced -C Runtime Array Checking:**

  In this `f95` compiler release, runtime array subscript range checking with the `-C` option has been enhanced to include array conformance checking. A runtime error is produced when an array syntax statement is executed where the array sections are not conformable. See Section , "-C" on page 3-13.

- **Introducing Fortran 2000 Features:**

  Some new formatted I/O features proposed for the next Fortran standard have been implemented in this release of `f95`. These are the `DECIMAL=`, `ROUND=`, and `IOMSG=` specifiers, and they may appear in `OPEN`, `READ`, `WRITE`, `PRINT`, and `INQUIRE` statements. Also implemented are the `DP`, `DC`, `RP`, and `RC` edit descriptors. See Section 4.6.9, "Fortran 2003 Formatted I/O Features" on page 4-18 for details.

- **Rounding in Formatted I/O:**

  A new option flag, `-iorounding`, sets the default rounding mode for formatted I/O. The modes, processor-defined or compatible, correspond to the `ROUND=` specifier implemented as part of the Fortran 2000 features. See Section , "-iorounding[={compatible|processor-defined}]" on page 3-32.

- **Obsolete Flags Removed:**

  The following flags have been removed from the `f95` command line:

  ```
  -db  -dbl
  ```

  The following `f77` compiler flags have not been implemented in the `f95` compiler and are also considered obsolete:

  ```
  -arg=local -i2 -i4 -misalign -oldldo -r8 -vax
  -xl -xvpara -xtypemap=integer:mixed
  ```

- **Checking for Stack Overflow:**

  Compiling with the new `-xcheck=stkovf` flag adds a runtime check for stack overflow conditions on entry to subprograms. If a stack overflow is detected, a `SIGSEGV` segment fault is raised. Stack overflows in multithreaded applications with large arrays allocated on the stack can cause silent data corruption in neighboring thread stacks. Compile all routines with `-xcheck=stkovf` if stack overflow is suspected. See Section , "-xcheck=keyword" on page 3-63.

- **New Default Thread Stack Size:**

  With this release, the default slave thread stack size has been increased to 4 Megabytes on SPARC V8 platforms, and 8 Megabytes on SPARC V9 platforms. See the discussion of stacks and stack sizes in the Parallelization chapter of the *Fortran Programming Guide* for details.

- **Enhanced Interprocedural Optimizations:**

  With `-xipo=1` the compiler does inlining across all source files. This release adds `-xipo=2` for enhanced interprocedural aliasing analysis and memory allocation and layout optimizations to imporve cache performance. See Section , "-xipo[={0|1|2}]" on page 3-74.

- **Control Prefetch Instructions With -xprefetch_level=n:**

  Use the new flag `-xprefetch_level=`*n* to control the automatic insertion of prefetch instructions with `-xprefetch=auto`. Use requires an optimization level of `-xO3` or greater and a target platform that supports prefetch (`-xarch` platforms `v8plus`, `v8plusa`, `v8plusb`, `v9`, `v9a`, `v9b`, `generic64`, or `native64`). See Section , "-xprefetch_level={1|2|3}" on page 3-87.

Feature histories for releases prior to Forte Developer 7 can be found in the documentation sets for those earlier releases on the `http://docs.sun.com` web site.

APPENDIX **C**

Legacy –xtarget Platform Expansions

This Appendix details older and less commonly used `-xtarget` option platform system names and their expansions. They appear here for reference purposes. The values for UltraSPARC platforms are given under the `-xtarget` option description in Chapter 3. Some of the system platforms listed here may no longer be supported by recent releases of the Solaris operating environment.

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 C-1 Legacy `-xtarget` Expansions

-xtarget=	-xarch	-xchip	-xcache
cs6400	v8	super	16/32/4:2048/64/1
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

TABLE C-1 Legacy `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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

TABLE C-1 Legacy `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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

TABLE C-1 Legacy `-xtarget` Expansions *(Continued)*

-xtarget=	-xarch	-xchip	-xcache
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
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

APPENDIX **D**

Fortran Directives Summary

This appendix summarizes the directives recognized by `f95` Fortran compiler:

- General Fortran Directives
- Sun Parallelization Directives
- Cray Parallelization Directives
- OpenMP Fortran 95 Directives, Library Routines, and Environment

D.1 General Fortran Directives

General directives accepted by `f95` are described in Chapter 2.

TABLE D-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$PRAGMA C (`*list*`)` Declares a list of names of external functions as C language routines.
`IGNORE_TKR` *Directive*	`C$PRAGMA IGNORE_TKR` {*name* {, *name*} ...} The compiler ignores the type, kind, and rank of the specified dummy argument names appearing in a generic procedure interface when resolving a specific call.

TABLE D-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.)

`ASSUME` *Directives*	`C$PRAGMA [BEGIN] ASSUME (`*expression* `[,`*probability*`])`
	`C$PRAGMA END ASSUME`
	Make assertions about conditions at certain points in the program that the compiler can assume are true.

D.2 Special Fortran 95 Directives

The following directives are only available with `f95`. See Section 4.8.2, "`FIXED` and `FREE` Directives" on page 4-22 for details.

TABLE D-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.

D.3 Fortran 95 OpenMP Directives

The Sun Fortran 95 compiler supports the OpenMP 2.0 Fortran API. The `-openmp` compiler flag enables these directives. (See Section , "`-openmp[={parallel|noopt|none}]`" on page 3-41).

See the *OpenMP API User's Guide* for complete details.

D.4 Sun Parallelization Directives

OpenMP parallelization is the preferred parallelization model with Fortran 95. Sun-style parallelization directives are described here for legacy applications, and are detailed in the chapter on parallelization in the *Fortran Programming Guide*.

TABLE D-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$PAR TASKCOMMON` *block_name* Declares variables in common block *block_name* as thread-private: private to a thread, but global within the thread. Declaring a common block `TASKCOMMON` requires that this directive appear after *every* common declaration of that block.
`DOALL` *Directive*	`C$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.

D.5 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 D-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.

Index

D

E

F

G

H

I

L

M

P

R

S

T

U

V

W