Oracle® Solaris Studio 12.4: Fortran User's Guide

ORACLE®

Contents

Using This Documentation

- **Overview** – Describes the Oracle Solaris Studio Fortran compiler and environment
- **Audience** – Application developers, system developers, architects, support engineers
- **Required knowledge** – Programming experience, software development testing, aptitude to build and compile software products

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://docs.oracle.com/cd/E37069_01.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

1

Introduction

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

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

1.1   Standards Conformance

- `f95` conforms to part one of the ISO/IEC 1539-1:1997 Fortran standards document.
- Floating-point arithmetic is based on IEEE standard 754-1985, and international standard IEC 60559:1989.
- See Chapter 4, "Solaris Studio Fortran Features and Extensions" for a list of Fortran 2003 and Fortran 2008 features and other language extensions implemented in this release.
- `f95` provides support for the optimization-exploiting features of the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium Pro, and Xeon Intel®64, on Solaris and Linux (x86) platforms.
- Oracle Solaris Studio compilers support the OpenMP API forshared memory parallelism, Version 4.0. See the *OpenMP API User's Guide* for details.
- In this document, "Standard" means conforming to the versions of the standards listed above. "Non-standard" or "Extension" refers to features that go beyond these versions of these standards.

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

1.2   Features of the Fortran Compiler

The Oracle Solaris Studio Fortran compiler, `f95`, provides the following features and extensions:

- Global program checking across routines for consistency of arguments, commons, parameters, and the like. (Only available on Solaris platforms).
- Optimized automatic and explicit loop parallelization for multiprocessor systems.
- VAX/VMS Fortran extensions.
- OpenMP parallelization directives.
- Global, peephole, and potential parallelization optimizations produce high performance applications. Benchmarks show that optimized applications can run significantly faster when compared to unoptimized code.
- Common calling conventions permit routines written in C or C++ to be combined with Fortran programs.
- Support for 64-bit enabled Solaris and Linux environments.
- IEEE quadruple precision REAL and COMPLEX support on all platforms.
- Compatibility between Fortran 77 and Fortran 95/Fortran 2003 programs and object binaries.
- Interval Arithmetic programming.
- Support for unsigned integer types.
- Many Fortran 2003 and Fortran 2008 features.

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

See also the *What's New In Oracle Solaris Studio 12.4* guide for latest information about new and changed features in this release of the compilers and tools, as well as known issues, workarounds, and limitations. The *What's New* guide can be accessed from the documentation index for this release at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`

1.3   Other Fortran Utilities

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

- **Solaris 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. 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. (Solaris platforms only)
- **Solaris Studio** dbx—Provides a robust and feature-rich runtime and static debugger, and includes a performance data collector.

1.5   Sun Performance Library

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

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 *Sun Performance Library User's Guide* for details. (Man pages for the performance library routines are in section 3P.)

1.6   Interval Arithmetic

The f95 compiler provides the compiler flags -xia and -xinterval to enable new language extensions and generate the appropriate code to implement interval arithmetic computations. See the *Fortran 95 Interval Arithmetic Programming Reference* for details.

1.7   Man Pages

Online manual (man) pages provide immediate documentation about a command, function, subroutine, or collection of such things. The user's MANPATH environment variable should be set to the path to the installed Solaris Studio man directory to access the Oracle Solaris 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)	Solaris Studio Performance Analyzer
asa(1)	Fortran carriage-control print output post-processor
dbx(1)	Command-line interactive debugger
fpp(1)	Fortran source code preprocessor
cpp(1)	C source code preprocessor
fdumpmod(1)	Display contents of a MODULE (.mod) file.
fsplit(1)	Preprocessor 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
ld(1)	Link editor for object files

1.8   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.
```

2

♦♦♦ **C H A P T E R   2**

Using Solaris Studio Fortran

This chapter describes how to use the Fortran 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 compiler can also be used to:

- Generate a parallelized executable file for execution by multiple threads (`-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 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`, `.F95`, `.f03`, or `.F03` filename suffix; (see Table Table 2-1), 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 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.

, 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 Compiler

Suffix	Language	Action
`.f`	Fortran77 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 fixed-format	Same action as `.f`.

Suffix	Language	Action
`.F`	Fortran 77 or Fortran 95 fixed-format	Apply the Fortran (or C) preprocessor to the Fortran 77 source file before compilation.
`.F95` `.F90`	Fortran 95 free-format	Apply the Fortran (or C) preprocessor to the Fortran 95 free-format source file before Fortran compiles it.
`.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, .so,.` `.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 "4.1 Source Language Features" on page 161.

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 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 Netlib.

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   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 "4.9 Module Files" on page 184.

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 recognizes a set of general directives and parallelization directives, including OpenMP directives.

Directives unique to `f95` are described in "4.8 Directives" on page 182. A complete summary of all the directives recognized by `f95` appears in Appendix C, "Fortran Directives Summary".

Note - Directives are not part of the Fortran standard.

2.3.1   General Directives

The various forms of a general Fortran directive are:

`!$PRAGMA` *keyword* (*a* [, *a*] …) [, *keyword* (*a* [, *a*] …)] ,…

`!$PRAGMA  SUN` *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`, 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 (`!$PRAGMA`). The examples in this chapter assume Fortran 95 free-format.
- The next seven characters are `$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 `!$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:

  `!$PRAGMA Sun UNROLL=3.`

  Another workaround might be to add `-Usun` when compiling a `.F` file.

The Fortran compiler recognizes the following general directives:

TABLE 2-2   Summary of General Fortran Directives

C *Directive*	!$PRAGMA C(*list*)
	Declares a list of names of external functions as C language routines.
IGNORE_TKR *Directive*	!$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*	!$PRAGMA SUN UNROLL=*n*
	Advises the compiler that the following loop can be unrolled to a length *n*.
WEAK *Directive*	!$PRAGMA WEAK(*name*[=*name2*])
	Declares *name* to be a weak symbol, or an alias for *name2*.
OPT *Directive*	!$PRAGMA SUN OPT=*n*
	Set optimization level for a subprogram to *n*.
PIPELOOP *Directive*	!$PRAGMA SUN PIPELOOP=*n*
	Assert dependency in the following loop exists between iterations *n* apart.
PREFETCH *Directives*	!$PRAGMA SUN_PREFETCH_READ_ONCE(*name*) !$PRAGMA SUN_PREFETCH_READ_MANY(*name*) !$PRAGMA SUN_PREFETCH_WRITE_ONCE(*name*) !$PRAGMA SUN_PREFETCH_WRITE_MANY(*name*)
	Request compiler generate prefetch instructions for references to name. (Requires -xprefetch option, which is enabled by default. Prefetch directives can be disabled by compiling with –xprefetch=no. Target architecture must also support prefetch instructions, and the compiler optimization level must be greater than –xO2.)
ASSUME *Directives*	!$PRAGMA [BEGIN] ASSUME (*expression* [,*probability*]) !$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
!$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 `!$PRAGMA`.

The `!$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:

```
!$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.

```
!$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 `!$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:

```
!$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 **PIPELOOP[=***n***]** Directive

The PIPELOOP=*n* directive requires that you specify SUN after !$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. (Default if *n* is not specified is 0)

```
C    We know that the value of K is such that there can be no
C    cross-iteration dependencies (E.g. K>N)
!$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.*

2.3.1.7 The **PREFETCH** Directives

The -xprefetch option flag, "3.4.166 –xprefetch[=*a*[,*a*]]" on page 141, enables a set of PREFETCH directives that advise the compiler to generate prefetch instructions for the specified data element on processors that support prefetch.

```
!$PRAGMA SUN_PREFETCH_READ_ONCE(name)
!$PRAGMA SUN_PREFETCH_READ_MANY(name)
!$PRAGMA SUN_PREFETCH_WRITE_ONCE(name)
!$PRAGMA SUN_PREFETCH_WRITE_MANY(name)
```

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

2.3.1.8   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

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

Alternatively, the "range assertion" `ASSUME` is:

```
!$PRAGMA BEGIN ASSUME [expression [, probability)
      block of statements
!$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.

```
!$PRAGMA BEGIN ASSUME(__tripcount().GE.10000,1) !! a big loop
      do i = j, n
         a(i) = a(j) + 1
      end do
!$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 "3.4.110 – xassume_control[=*keywords*]" on page 105) 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:

```
!$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

OpenMP parallelization directives are only recognized when compiling with `-openmp`. Details regarding OpenMP parallelization can be found in the *OpenMP API User's Guide*.

The Fortran compiler supports the OpenMP API for shared memory parallelism, Version 4.0. Legacy Sun and Cray parallelization directives are now deprecated and should not be used.

2.3.2.1 OpenMP Parallelization Directives

The Fortran compiler recognizes the OpenMP API for shared memory parallelism 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 `-xopenmp`, to enable OpenMP directives. (see "3.4.157 –xopenmp[={parallel|noopt|none}]" on page 138.)

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

Note - Legacy Sun and Cray style parallelization directives are now deprecated. The OpenMP parallelization API is preferred.

2.3.3   `IVDEP` Directive

The `!DIR$ IVDEP` directive tells the compiler to ignore some or all loop-carried dependences on array references that it finds in a loop, allowing it to perform various loop optimizations such as microvectorization, distribution, software pipelining, among others, that would not be otherwise possible. It is employed in situations where the user knows either that the dependences do not matter or that they never occur in practice.

For example:

```
DO I = 1, N
  A(V(I)) = A(V(I)) + C(I)
END DO
```

In this loop there are several loop-carried dependences on `A(V(I))`, because `V(I)` may contain duplicate values to index `A`, and reordering the loop might result in different results. But if it is known that `V` contains only distinct values, the loop could be reordered safely, and an `IVDEP` directive could be used to allow optimization.

The –`xivdep` compiler option (see "3.4.136 –`xivdep`[=*p*]" on page 128) can be used to disable or determine the interpretation of `IVDEP` directives.

Some legacy interpretations of the `IVDEP` directive only assert that there are no *backward loop-carried dependences*. The Fortran compiler's default is –`xivdep=loop`, indicating that the `IVDEP` directive asserts there are no assumed loop dependences.

The following examples illustrate backward and forward dependences.

```
 do i = 1, n   ! BACKWARD LOOP-CARRIED DEPENDENCE
... = a(i-1)   ! S1
   a(i) = ...   ! S2
end do

do i = 1, n   ! FORWARD LOOP-CARRIED DEPENDENCE
   a(i) = ...   ! S3
... = a(i-1)  ! S4
end do
```

The first loop has a backward loop-carried dependence from S2 to S1, and the second loop has a forward loop-carried dependence from S3 to S4. Since the second loop has only a forward dependence, it can be distributed and/or microvectorized safely while the first one cannot.

Here is an example of the use of IVDEP under the default value of `-xivdep=loop`:

```
integer target a(n)
integer, pointer :: p(:), q(:)
!DIR$ IVDEP
do i = 1, n
  p(i) = q(i)
  a(i) = a(i-1)
end do
```

The assumed dependences between `p(i)` and `q(i)` and between `p(i)` and `a(*)` are ignored, but the obvious dependences between `a(i)` and `a(i-1)` are not. The loop can be divided into two loops and the resulting `p(i) = q(i)` loop can be microvectorized.

The IVDEP directive applies to the immediately following DO loop. No other code is allowed between the directive and the loop. `!DIR$ IVDEP` can also be applied to an array assignment, FORALL, or WHERE construct. If multiple directives are present for a particular loop (such as IVDEP and UNROLL), the compiler will obey all of them, if possible.

2.4 Library Interfaces and `system.inc`

The Fortran 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 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 compiler efficiently. A complete compiler options reference follows in the next chapter.

2.5.1   Determining Hardware Platform

Some compiler flags allow the user to tune code generation to a specific set of hardware platform options. The compiler's `-dryrun` option can be used to determine the native processor:

```
<sparc>% f95 -dryrun -xtarget=native
###     command line files and options (expanded):
### -dryrun -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 -xchip=ultra3i

<x64>% f95 -dryrun -xtarget=native
###     command line files and options (expanded):
### -dryrun -xarch=sse2a -xcache=64/64/2:1024/64/16 -xchip=opteron
```

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 Oracle Solaris Studio 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` (Module files). See the *Fortran Programming Guide* 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. 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 Oracle Solaris operating system command, `swap -s`, displays available swap space. See `swap`(1M).

Example: Use the `swap` command:

```
demo% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used, 1058708k available
To determine the actual real memory:

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

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

2.6   User-Supplied Default Options File

The default compiler options file enables the user to specify a set of default options that are applied to all compiles, unless otherwise overridden. For example, the file could specify that all compiles default at –xO2, or automatically include the file `setup.il`.

At startup, the compiler searches for a default options file listing default options it should include for all compiles. The environment variable `SPRO_DEFAULTS_PATH` specifies a colon separated list of directories to search for the the defaults file.

If the environment variable is not set, a standard set of defaults is used. If the environment variable is set but is empty, no defaults are used.

The defaults file name must be of the form compiler.defaults, where compiler is one of the following: `cc`, `c89`, `c99`, `CC`, `ftn`, or `lint`. For example, the defaults for the Fortran compiler would be `ftn.defaults`

If a defaults file for the compiler is found in the directories listed in `SPRO_DEFAULTS_PATH`, the compiler will read the file and process the options prior to processing the options on the command line. The first defaults file found will be used and the search terminated.

System administrators may create system-wide default files in *Studio-install-path*`/lib/` `compilers/etc/config`. If the environment variable is set, the installed defaults file will not be read.

The format of a defaults file is similar to the command line. Each line of the file may contain one or more compiler options separated by white space. Shell expansions, such as wild cards and substitutions, will not be applied to the options in the defaults file.

The value of the `SPRO_DEFAULTS_PATH` and the fully expanded command line will be displayed in the verbose output produced by the –`dryrun` option.

Options specified by the user on the command line will usually override options read from the defaults file. For example, if the defaults file specifies compiling with –x04 and the user specifies –x02 on the command line, –x02 will be used.

Some options appearing in the default options file will be appended after the options specified on the command line. These are the preprocessor option –I, linker options –B, –L, –R, and –l, and all file arguments, such as source files, object files, archives, and shared objects.

The following is an example of how a user-supplied default compiler option startup file might be used.

```
demo% cat /project/defaults/ftn.defaults
-I/project/src/hdrs —L/project/libs —lliblproj —xvpara
demo% setenv SPRO_DEFAULTS_PATH  /project/defaults
demo% f95 —c —I/local/hdrs —L/local/libs —lliblocal tst.f
```

This command is now equivalent to:

```
f95 -fast —xvpara —c —I/local/hdrs —L/local/libs —lliblocal tst.f \
    —I/project/src/hdrs —L/project/libs —lliblproj
```

While the compiler defaults file provides a convenient way to set the defaults for an entire project, it can become the cause of hard to diagnose problems. Set the environment variable SPRO_DEFAULTS_PATH to an absolute path rather than the current directory to avoid such problems.

The interface stability of the default options file is uncommitted. The order of option processing is subject to change in a future release.

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 "3.1 Command Syntax" on page 43.
- Summaries of options arranged by functionality starts at "3.3 Options Summary" on page 45.
- The complete reference detailing each compiler option flag starts at "3.4 Options Reference" on page 51.

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

Syntax Format	Example
–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 x64/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`
Do not replace a module file whose content is not changed by the compilation of the associated Fortran module	`-keepmod`
Retains the temporary files that are created during compilation.	`-keeptmp`
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*
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*
Generate make dependencies	`-xM`
(Oracle Solaris) Link debug information from object files into executable	`-xs`
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`

Function	Option Flag
(x86) Save copies of register-based function arguments in the stack	`-preserve_argvalues`
STOP returns integer status value to shell	`-stop_status[=`*yn*`]`
Specify code address space	`-xcode=`*x*
Equivalent to `-inline`	`-xinline`
Manually change the heuristics used by the compiler for deciding when to inline a function call	`-xinline_param`
Generate a report written to standard output on the inlining of functions by the compiler	`-xinline_report`
Enable 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`
Support file sharing between little-endian and big-endian platforms.	`-xfilebyteorder`
Specify memory alignment and behavior	`-xmemalign[=`*ab*`]`

Debugging:

Enable runtime subscript range checking	`-C`
Compile for debugging with `dbx`	`-g`
Flag use of undeclared variables	`-u`
Check `!$PRAGMA ASSUME` assertions	`-xassume_control=check`
Check for stack overflow at runtime	`-xcheck=stkovf`
Enable runtime task common check	`-xcommonchk`
Control how much debugging and observability information is emitted	`-xdebuginfo`
Compile for Performance Analyzer	`-xF`
Control globalization of file static variables but not functions	`-xglobalize`
Keep definitions of unreferenced functions and variables	`-xkeep_unref`
Generate cross-reference listings	`-Xlist`*x*
Enable debugging without object files	`-xs`

Diagnostics:

Flag use of non-standard extensions	`-ansi`
Suppress named warning messages	`-erroff=`
Display error tag names with error messages	`-errtags`
Show summary of compiler options	`-flags`, `-help`

Function	Option Flag
Show version number of the compiler and its phases	`-V`
Verbose messages	`-v`
Verbose parallelization messages	`-vpara`
Show/suppress warning messages	`-w`*n*
(Oracle Solaris) Link debug information from object files into executable	`-xs`
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*
Do not build library search path into executable.	`-norunpath`
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`
Reserve an area of memory before the start of each function	`-xpatchpadding`
Specify whether the program contains references to dynamically bound symbols	`-xunboundsym`
Link editor option	`-z`*x*
Generate pure libraries with no relocations	`-ztext`
Numerics and Floating-Point:	
Use non-standard floating-point preferences	`-fnonstd`
Select non-standard floating point	`-fns`
Enable runtime floating-point overflow during input	`-fpover`
Select IEEE floating-point rounding mode	`-fround=`*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`

Function	Option Flag
Optimize using a selection of options	`-fast`
Specify that the program does not perform I/O in more than one thread at a time.	`-fserialio`
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*`]`
Invoke interprocedural optimizations pass	`-xipo[=`*n*`]`
Reduce compile time by avoiding optimizations during the initial pass through the compiler, optimizing only at link time	`-xipo_build`
Set highest optimization level for `#pragma OPT`	`-xmaxopt[=`*n*`]`
Compile for post-compilation optimizations	`-xbinopt=prepare`
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`
Cause the driver to include a special mapfile on the link line.	`-xsegment_align`
Do no optimizations that increase code size	`-xspace`
Specify that the application will be run in situations where many processes are simultaneously running on the system	`-xthroughput`
Generate calls to vector library functions automatically	`-xvector[=`*yn*`]`

Parallelization:

Function	Option Flag
Enable automatic parallelization of DO loops	`-autopar`
Equivalent to `-xopenmp=parallel`	`-fopenmp`
Show loop parallelization information	`-loopinfo`
Compile for hand-coded multithreaded programming	`-mt`
Accept OpenMP API directives. The -xopenmp option flag accepts the following suboption keywords: `parallel`, `noopt`, and `none`.	`-xopenmp[=`*keyword*`]`
Recognize reduction operations in loops with automatic parallelization	`-reduction`
Verbose parallelization messages	`-vpara`
Specify that the application will be run in situations where many processes are simultaneously running on the system	`-xthroughput`

Source Code:

Function	Option Flag
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`

Function	Option Flag
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 memory model, 32 or 64 bits.	`-m32	—m64`
Specify target platform instruction set for the optimizer	`-xarch=`*a*
Specify target cache properties for optimizer	`-xcache=`*a*
Specify target processor for the optimizer	`-xchip=`*a*
Specify target platform for the optimizer	`-xtarget=`*a*

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	See description of -fast for complete current expansion.
-fnonstd	-fns -ftrap=common
-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 or add to the the expansion of the macro.

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
Preserve actual arguments over ENTRY statements	-arg=local
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
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 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`
`-native`	`-xtarget=native`
`-noqueue`	License queueing. No longer needed.
`-p`	Profiling. Use `-pg` or the Performance Analyzer
`-pic`	`-xcode=pic13`
`-PIC`	`-xcode=pic32`
`-silent`	Ignored.
`-xarch={v7,v8,v8a}`	Use `–m32`

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 x64/x86 platforms. Option flags valid only on SPARC platforms are marked **(SPARC)**. Option flags valid only on x64/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.

3.4.1   `–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.

Do not use `-aligncommon=1` with `-xmemalign` as these declarations will conflict and could cause a segmentation fault on some platforms and configurations.

Using `–aligncommon=1` on SPARC platforms might result in a bus error due to misalignment, requiring an appropriate choice of the `–xmemalign` option be used. Depending on the application, `–xmemalign=1s`, `–xmemalign=4i`, or `–xmemalign=8i` should give optimal performance while avoiding the segmentation fault.

See also `–xmemalign`

3.4.2   `–ansi`

Identify many nonstandard extensions.

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

3.4.3   `–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.

3.4.4   —`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. The -stackvar option may provide better performance when using -autopar because it may allow the optimizer to detect additional opportunities for parallelization. See the description of the -stackvar option for information on how to set the sizes for the main thread stack and for the slave thread stacks.

Avoid -autopar if the program already contains explicit calls to the libthread threads library. See note in "3.4.60 -mt[={yes|no}]" on page 80.

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

Use the OMP_NUM_THREADS environment variable to specify the number of threads to use when running a program automatically parallelized by the -xautopar compiler option. If OMP_NUM_THREADS is not set, the default number of threads used is equal to the number of cores on the machine, capped at 32. Set OMP_NUM_THREADS to 1 to run with just one thread. For best performance, the number of threads used should not exceed the number of hardware threads (or virtual processors) available on the machine. On Oracle Solaris systems, this number can be determined by using the psrinfo(1M) command. On Linux systems, this number can be determined by inspecting the file /proc/cpuinfo. See the *OpenMP API User's Guide* for more information.

In addition to OMP_NUM_THREADS, other environment variables that apply to an OpenMP programs may be used with a program automatically parallelized by the -xautopar compiler option. See the *Oracle Solaris Studio OpenMP API User's Guide* for descriptions of the environment variables.

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.

Use the -reduction option in conjunction with -autopar to recognize reduction operations in loops.

Use the -loopinfo option to show which loops were and were not parallelized.

For explicit, user-controlled parallelization, use OpenMP directives and the -xopenmp option.

3.4.5   `—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 `-B`*x* on the compile command will require it in the link step as well.

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

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

Mixing static Fortran runtime system libraries with dynamic Fortran runtime system libraries is not recommended and can result in linker errors or silent data corruption. Always link with the latest shared dynamic Fortran runtime system libraries.

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

3.4.6   `—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.

3.4.7   **–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.

3.4.8   **–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.

3.4.9   **–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 appeared 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 compiler version is predefined (in hex) in `_ _SUNPRO_F90`, and `_ _SUNPRO_F95`. For example `_ _SUNPRO_F95` is `0x860` for version 8.6 of the Fortran compiler in the Oracle Solaris Studio 12.4 release.
- The following macros are predefined on appropriate systems:

  `_ _sparc, _ _unix, _ _sun, _ _SVR4, __i386, _ _SunOS_5_10, _ _SunOS_5_11`

  The symbols `_ _sparc, _ _sparcv8`, and `_ _sparcv9` are defined on their respective SPARC systems.
- The following are predefined with no underscores, but they might be deleted in a future release: `sparc, unix, sun`
- On 64-bit x86 systems, the macros `__amd64` and `__x86_64` are defined.

Compile a `.F`, `.F90`, `.F95`, or `.F03` source file with the `-v` verbose option to see the preprocessor definitions assumed 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`.

3.4.10　`–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 `-m64`). 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.

Using `-dalign` along with `-xtypemap=real:64,double:64,integer:64` also causes 64-bit integer variables to be double-word aligned on SPARC processors.

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 "3.4.1 —aligncommon[={1|2|4|8|16}]" on page 51

3.4.11   —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 `-m64`, 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.

3.4.12   –depend[={yes|no}]

Analyze loops for inter-iteration data dependencies and performs loop restructuring. Loop restructuring includes loop interchange, loop fusion, and scalar replacement.

If you do not specify -depend, the default is -depend=yes. If you specify -depend but do not specify an argument, the compiler assumes -depend=yes.

To turn off dependence analysis, compile with -depend=no.

-xdepend is a synonym for -depend.

3.4.13   –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.

3.4.14   –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.

3.4.15   `–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.

3.4.16   `–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. `–erroff` without an argument is equivalent to `–erroff=%all`.

Example:

```
f95 -erroff=WDECL_LOCAL_NOTUSED ink.f
```

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

3.4.17   `–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. `-errtags` alone is equivalent to `-errtags=yes`.

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)
```

3.4.18 **–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`.

3.4.19 **–ext_names=***e*

Create external names with or without trailing underscores.

e must be either `plain`, `underscores`, or `fsecond-underscore` The default is `underscores`.

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

`–ext_names=underscores`: Add trailing underscore.

`–ext_names=fsecond-underscore`: Append two underscores to external names that contain an underscore, and a single underscore to those that do not.

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 routines to call (and be called by) other programming language routines.

`fsecond-underscore` is provided for compatibility with `gfortran`.

3.4.20 **–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.

3.4.21   –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 environments with -m64).

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 "3.4.10 –dalign" on page 56. Because -dalign is part of the -fast option, so is -f.

3.4.22   –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 Sun WorkShop f77 compiler, to the f95 Fortran compiler. (There is no longer a separate FORTRAN 77 compiler.)

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

keyword	meaning
%all	Enable all the Fortran 77 compatibility features.

keyword	meaning
`%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.
`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 trapping mode, which is `-ftrap=common`. `f95` 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 "4.12 Mixing Languages" on page 187 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.

3.4.23 `—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 -dryrun flag to see the expansion of -fast.

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

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

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

–fast selects the following options:

- 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=ultraT2 ...
- The -O5 optimization level option.
- The -depend option analyzes loops for data dependencies and possible restructuring. (This option is always enabled when compiling at optimizations levels -xO3 and greater.)
- 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 "3.4.36 –fsimple[={1|2|0}]" on page 69.
- 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=lib transforms certain math library calls within DO loops to single calls to a vectorized library equivalent routine with vector arguments. (SPARC)

- `-fma=fused` enables automatic generation of floating-point fused multiply-add instructions.
- `–fns` selects non-standard floating-point arithmetic exception handling and gradual underflow. See "3.4.28 `–fns[={yes|no}]`" on page 66.
- `-fround=nearest` is selected because `–xvector` and `–xlibmopt` require it. (Oracle Solaris)
- Trapping on common floating-point exceptions, `-ftrap=common`, is the enabled with `f95`.
- `-nofstore` cancels forcing expressions to have the precision of the result. (x86)
- `-xregs=frameptr` on x86 allows the compiler to use the frame-pointer register as a general purpose register. See the description of `–xregs=frameptr` for details and especially if compiling mixed C, C++, and Fortran source code. Specify `-xregs=no%frameptr` after `-fast` and the frame pointer register will not be used as a general purpose register. (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. Invoking the compiler with `-dryrun` displays the `-fast` expansion:

```
<sparc>% f95 -dryrun -fast |& grep ###
        ###     command line files and options (expanded):
        ### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4
            -xchip=ultra3i -xdepend=yes -xpad=local -xvector=lib
            -dalign -fsimple=2 -fns=yes -ftrap=common -xlibmil
            -xlibmopt -fround=nearest
```

3.4.24 `–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.

3.4.25 `–flags`

Synonym for `-help`.

3.4.26 `—fma[={none|fused}]`

Enables automatic generation of floating-point fused multiply-add instructions. `-fma=none` disables generation of these instructions. `-fma=fused` allows the compiler to attempt to find opportunities to improve the performance of the code by using floating-point fused multiply-add instructions.

The default is `-fma=none`.

The minimum architecture requirement is `-xarch=sparcfmaf` on SPARC and `-xarch=avx2` on x86 to generate fused multiply-add instructions. The compiler marks the binary program if fused multiply-add instructions are generated to prevent execution of the program on platforms that do not support fused multiply-add instructions. When the minimum architecture is not used, then `-fma=fused` has no effect.

Fused multiply-add instructions eliminate the intermediate rounding step between the multiply and add. Consequently, programs may produce different results when compiled with `-fma=fused`, although precision will tend to increase rather than decrease.

3.4.27 `—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*.

3.4.28   –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).

On x86, `-fns` selects SSE flush-to-zero mode and where available, denormals-are-zero mode. This flag causes subnormal results to be flushed to zero. Where available, this flag also causes subnormal operands to be treated as zero. This flag has no effect on traditional x87 floating-point operations not utilizing the SSE or SSE2 instruction set.

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

3.4.29 `-fopenmp`

Same as `-xopenmp=parallel`.

3.4.30 `–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`. Combine with `–ftrap` to get full diagnostic information.

3.4.31 `–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 "3.4.165 `–xpp={fpp|cpp}`" on page 141.

3.4.32 `–fprecision={single|double|extended}`

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

Sets the floating-point precision mode to either `single`, `double`, or `extended` on x86 platforms.

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, but is ignored if compiling for 64–bit (`-m64`) or SSE2–enabled (`-xarch=sse2`) processors. It is also ignored on SPARC systems.

3.4.33   `—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.

3.4.34   `—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*.)

3.4.35   `—fserialio`

A linking option that specifies that the program does not perform I/O in more than one thread at a time. It allows Fortran I/O statements to be executed without performing synchronization

to avoid race conditions. This option should be specified only when creating an executable program. It should not be specified when creating a shared object library nor should it be specified if the program includes code compiled with versions of Sun f77 before the Sun Forte 7 release.

3.4.36   `−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.

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; for example, `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`	In addition to `−fsimple=1`, 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` selects `-fsimple=2`.

3.4.37   `—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.

3.4.38   `—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,division`.

The `f95` default is `-ftrap=common`. This differs from the C and C++ compiler defaults, `-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*.

3.4.39   **–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.

3.4.40   **-g**

See `-g[`*n*`]`.

3.4.41   **–g[**n**]**

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.

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 *Solaris 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.

-g is implemented as a macro that expands to various other, more primitive, options. See -xdebuginfo for the details of the expansions.

-g	Produce standard debugging information.
-gnone	Do not produce any debugging information. This is the default.
-g1	Produce file and line number as well as simple parameter information that is considered crucial during post-mortem debugging.
-g2	Same as -g.
-g3	Produce additional debugging information, which currently consists only of macro definition information. This added information can result in an increase in the size of the debug information in the resulting .o and executable when compared to using only -g.

3.4.42   **–h**_name_

Specify the name of the generated dynamic shared library.

This option is passed on to the linker. For details, see the Oracle 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.

3.4.43 **–help**

Display a summary list of compiler options.

See also "3.4.125 –`xhelp=flags`" on page 118.

3.4.44 **–I***path*

Add *path* to the `INCLUDE` file search path.

Insert the directory path *path* at the start of the `INCLUDE` file search path. A 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.

The search path is also used to find `MODULE` files.

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.

3.4.45   `-i8`

(*There is no* `-i8` *option.*)

Use `—xtypemap=integer:64` to specify 8–byte `INTEGER` with this compiler.

3.4.46   `—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`.

Specifying `-xinline=` without naming any functions or `%auto` indicates that none of the routines in the source files are to be inlined.

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.

3.4.47 `—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.

3.4.48 `—keepmod[={yes|no}]`

If a module file exists and its content is not changed by the latest compilation, it will not be replaced even though the compilation is supposed to create a new module file with the same name. Since the content of the module file is unchanged by the compilation, the only effect is that the time stamp of the existing module file will be preserved.

The default, when `-keepmod` is not specified, is `-keepmod=yes`. Note that this default is different from previous releases of Oracle Solaris Studio Fortran.

This option is best used together with the dependencies generated by the `-xM` compilation option. By retaining the time stamp of a module file when its content is unchanged, this option prevents cascading compilation for the source files depending on this module file. This is very helpful in an incremental build and can significantly reduce the build time.

When this option is used with user-specified dependencies and the user has an explicit build rule on how to create the module with a dependency on the corresponding source file, the

option can cause the source file to be recompiled multiple times even though the source file is modified only once because of the outdated time stamp of the module file.

3.4.49   **–keeptmp**

Retains the temporary files that are created during compilation.

3.4.50   **–Kpic**

(Obsolete) Synonym for `-pic`.

3.4.51   **–KPIC**

(Obsolete) Synonym for `-PIC`.

3.4.52   **–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 .

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
```

3.4.53 −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.

3.4.54 −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)

3.4.55 -library=sunperf

Link with the Oracle Solaris Studio supplied performance libraries. (See the *Sun Performance Library User's Guide*.)

3.4.56 −loopinfo

Show loop parallelization results.

Show which loops were and were not parallelized with the −autopar option.

−loopinfo displays a list of messages on standard error:

```
demo% f95 -c -fast -autopar -loopinfo shalow.f
...
"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
...
...etc
```

3.4.57   −M*path*

Specify MODULE directory, archive, or file.

Look in path for Fortran 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.

Directories named in −I *path* will be searched for module files if the files are not found in any of the other locations that are searched.

A space between the -M and the path is allowed. For example, -M /home/siri/PK15/Modules

On Solaris, if the path identifies a regular file that is not an archive or a module file, the compiler passes the option to the linker, `ld`, which will treat it as a linker mapfile. This feature is provided as a convenience similar to the C and C++ compilers.

See "4.9 Module Files" on page 184 for more information about modules in Fortran.

3.4.58   **—m32 | —m64**

Specify memory model for compiled binary object.

Use `-m32` to create 32-bit executables and shared libraries. Use `-m64` to create 64-bit executables and shared libraries.

The ILP32 memory model (32-bit `int`, `long`, `pointer` data types) is the default on all Solaris platforms and on Linux platforms that are not 64-bit enabled. The LP64 memory model (64-bit long, pointer data types) is the default on Linux platforms that are 64-bit enabled. `-m64` is permitted only on platforms that are enabled for the LP64 model.

Object files or libraries compiled with `-m32` cannot be linked with object files or libraries compiled with `-m64`.

When compiling applications with large amounts of static data on x64 platforms using `-m64, -xmodel=medium` may also be required.

Be aware that some Linux platforms do not support the medium model.

Note that in previous compiler releases, the memory model, ILP32 or LP64, was implied by the choice of the instruction set with `-xarch`. Starting with the Solaris Studio 12 compilers, this is no longer the case. On most platforms, just adding `-m64` to the command line is sufficient to create 64-bit objects.

On Solaris, `-m32` is the default. On Linux systems supporting 64-bit programs, `-m64 -xarch=sse2` is the default.

3.4.59   **—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 "4.9 Module Files" on page 184 for more information about modules in Fortran.

3.4.60 `-mt[={yes|no}]`

Use this option to compile and link multithreaded code using the Oracle Solaris threads or POSIX threads API. The `-mt=yes` option assures that libraries are linked in the appropriate order.

This option passes `-D_REENTRANT` to the preprocessor.

On Linux platforms, only the POSIX threads API is available. (There is no `libthread` on Linux platforms.) Consequently, `-mt=yes` on Linux platforms adds `-lpthread` instead of `-lthread`. To use POSIX threads on Linux platforms, compile with `-mt`.

Note that when compiling with `-G`, neither `-lthread` nor `-lpthread` are automatically included by `-mt=yes`. You will need to explicitly list these libraries when building a shared library.

The `-xopenmp` option (for using the OpenMP shared-memory parallelization API) includes `-mt=yes` automatically.

If you compile with `-mt=yes` and link in a separate step, you must use the `-mt=yes` option in the link step as well as the compile step. If you compile and link one translation unit with `-mt=yes`, you must compile and link all units of the program with `-mt=yes`

`-mt=yes` is the default behavior of the compiler. If this behavior is not desired, compile with `-mt=no`.

The option `-mt` is equivalent to `-mt=yes`.

3.4.61 `—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`.

3.4.62 `—noautopar`

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

3.4.63   **—nodepend**

Cancel any `-depend` appearing earlier on the command line. `-depend=no` is the preferred usage over `-nodepend`.

3.4.64   **-nofstore**

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

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

3.4.65   **—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.

3.4.66   **—nolibmil**

Cancel `-libmil` on command line.

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

```
demo% f95 -fast -nolibmil …
```

3.4.67   `—noreduction`

Disable `-reduction` on command line.

This option disables `-reduction`.

3.4.68   `—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.

3.4.69   `—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*.

3.4.70 **–o** *filename*

Specify the name of the executable file to be written.

There must be a blank between -o and *filename*. 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.

3.4.71 **—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.

3.4.72 **—openmp**

Synonym for `-xopenmp`.

3.4.73 **—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.

3.4.74 **—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 .

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.

3.4.75   –pg

Compile for profiling with the `gprof` profiler. (`-xpg` is a synonym for `-pg`)

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).

Note - There is no advantage compiling with `-xprofile` if you specify `-pg`. These two features do not prepare or use data provided by the other.

Profiles generated by using `prof`(1) or `gprof`(1) on 64 bit Solaris platforms or just `gprof` on 32 bit Solaris platforms include approximate user CPU times. These times are derived from PC sample data (see `pcsample`(2)) for routines in the main executable and routines in shared libraries specified as linker arguments when the executable is linked. Other shared libraries (libraries opened after process startup using `dlopen`(3DL)) are not profiled.

On 32 bit Solaris systems, profiles generated using `prof`(1) are limited to routines in the executable. 32 bit shared libraries can be profiled by linking the executable with `-pg` and using `gprof`(1).

The Solaris 10 software does not include system libraries compiled with `-p`. As a result, profiles collected on Solaris 10 platforms do not include call counts for system library routines.

The compiler options `-p, -pg`, or `-xpg` should not be used to compile multi-threaded programs, because the runtime support for these options is not thread-safe. If a program that uses multiple threads is compiled with these options invalid results or a segmentation fault could occur at runtime.

Binaries compiled with `-xpg` for `gprof` profiling should not be used with `binopt`(1) as they are incompatible and can result in internal errors.

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

On x86 systems, `-pg` is incompatible with `-xregs=frameptr`, and these two options should not be used together. Note also that `-xregs=frameptr` is included in `-fast`.

3.4.76 –`pic`

Compile position–independent code for shared library.

On SPARC, –`pic` is equivalent to `-xcode=pic13`. See for more information on position-indepented code.

On x86, produces position-independent code. Use this option to compile source files when building a shared library. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

3.4.77 **–PIC**

Compile position–independent code with 32-bit addresses.

On SPARC, –PIC is equivalent to -xcode=pic32. See "3.4.116 -xcode[=*v*]" on page 110 for more information about position-independent code.

On x86, –PIC is equivalent to –pic.

3.4.78 **–preserve_argvalues[=simple|none|complete]**

(x86) Saves copies of register-based function arguments in the stack.

When none is specified or if the -preserve_argvalues option is not specified on the command line, the compiler behaves as usual.

When simple is specified, up to six integer arguments are saved.

When complete is specified, the values of all function arguments in the stack trace are visible to the user in the proper order.

The values are not updated during the function lifetime on assignments to formal parameters.

3.4.79 **–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*.

3.4.80   **–qp**

Synonym for `-p`.

3.4.81   **–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 that is always searched last.

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

3.4.82   **–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 "3.4.183 –xtypemap=*spec*" on page 156.

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.

3.4.83 `—recl=a[,b]`

Set default output record length.

Set the default record length (in characters) for either or both preconnected units output (standard output) and error (standard error). This option must be specified using one of the following forms:

- `—recl=out:`*N*
- `—recl=error:`*N*
- `—recl=out:`*N1*`,error:`*N2*
- `—recl=error:`*N1*`,out:`*N2*
- `—recl=all:`*N*

where *N, N1, N2* are all positive integers in the range from 72 to 2147483646. `out` refers to standard output, `error` to standard error, and `all` sets the default record length to both. The default is `—recl=all:80`. This option is only effective if the program being compiled has a Fortran main program.

3.4.84 `—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 option `—autopar`. It is ignored otherwise. Explicitly parallelized loops are not analyzed for reduction operations.

3.4.85 `—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.

3.4.86 `–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`.

3.4.87 `–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.

3.4.88 `–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 slave thread of a multithreaded program has its own *thread* stack.

For slave threads, the default thread stack size is 4 Megabytes on 32–bit systems and 8 Megabytes on 64–bit systems. 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
```

You can set the size of the thread stack used by each slave thread by setting the STACKSIZE or OMP_STACKSIZE environment variable. See the *OpenMP API User's Guide* for more information about these environment variables.

Compile with -xcheck=stkovf to enable runtime checking for stack overflow situations. See the -xcheck option for more information.

3.4.89   **–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`.

3.4.90   `–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.

This option takes precedence over the value of the `TMPDIR` environment variable.

3.4.91   `–time`

Time each compilation phase.

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

3.4.92   `–traceback[={%none|common|`*signals_list*`}]`

Issue a stack trace if a severe error occurs in execution.

The `-traceback` option causes the executable to issue a stack trace to `stderr`, dump core, and exit if certain signals are generated by the program. If multiple threads generate a signal, a stack trace will only be produced for the first one.

To use traceback, add the `-traceback` option to the compiler command line when linking. The option is also accepted at compile-time but is ignored unless an executable binary is generated. Using `-traceback` with `-G` to create a shared library is an error.

TABLE 3-9   `-traceback` Options

Option	Meaning
`common`	specifies that a stack trace should be issued if any of a set of common signals occurs: `sigill`, `sigfpe`, `sigbus`, `sigsegv`, or `sigabrt`.
signals_list	specifies a comma-separated list of names of signals that should generate a stack trace, in lower case. The following signals (those that cause the generation of a core file) can be caught: `sigquit`, `sigill`, `sigtrap`, `sigabrt`, `sigemt`, `sigfpe`, `sigbus`, `sigsegv`, `sigsys`, `sigxcpu`, `sigxfsz`. Any of these can be preceded with `no%` to disable catching the signal. For example: `-traceback=sigsegv,sigfpe` will produce a stack trace and core dump if either `sigsegv` or `sigfpe` occurs.
`%none` or `none`	disables traceback

If the option is not specified, the default is `-traceback=%none`

`-traceback` alone, without a value, implies `-traceback=common`

Note: If the core dump is not wanted, users may set the `coredumpsize` limit to zero using:

`% limit coredumpsize 0`

The `-traceback` option has no effect on runtime performance.

3.4.93   **–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. Calls to intrinsic functions are not affected by this option.

Portability and mixing Fortran with other languages may require use of `-U`. See the *Fortran Programming Guide* chapter on porting programs to Solaris Studio Fortran.

3.4.94   **–U***name*

Undefine preprocessor macro *name*.

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

3.4.95 `–u`

Report undeclared variables.

Make the default type for all variables be *undeclared* rather than using Fortran implicit typing, as if `IMPLICIT NONE` appeared in each compilation unit. This option warns of undeclared variables, and does not override any `IMPLICIT` statements or explicit *type* statements.

3.4.96 `–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 "2.3.1.3 The `UNROLL` Directive" on page 31.

3.4.97 `–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 "4.9 Module Files" on page 184 for more information about modules in Fortran.

3.4.98   **–V**

Show name and version of each compiler pass.

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

3.4.99   **–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.

3.4.100   **–vax=***keywords*

Specify choice of legacy 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, and should be used with `–xmemalign` to avoid such errors.
`%all`	Enable all these VAX VMS features.
`%none`	Disable all these VAX VMS features.

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

Example:

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

The default is `-vax=%none`. Specifying `-vax` without any suboptions is equivalent to `-vax=%all`.

3.4.101   `—vpara`

Show parallelization warning messages.

Issues warnings about potential parallel programming problems in OpenMP programs.

Use with the `-xopenmp` option.

The compiler issues warnings when it detects the following situations:

- Loops are parallelized using OpenMP directives when there are data dependencies between different loop iterations.
- OpenMP data-sharing attributes-clauses are problematic. For example, declaring a variable "shared" whose accesses in an OpenMP parallel region may cause data race, or declaring a variable "private" whose value in a parallel region is used after the parallel region.

No warnings appear if all parallelization directives are processed without problems.

3.4.102   `-W`*c*`,`*arg*

Passes the argument *arg* to a specified component *c*.

Arguments must be separated from the preceding only by a comma. All `-W` arguments are passed after the rest of the command-line arguments. To include a comma as part of an argument, use the escape character `\` (backslash) immediately before the comma. All `-W` arguments are passed after the regular command-line arguments.

For example, `-Wa,-o,objfile` passes `-o` and `objfile` to the assembler in that order. Also, `-Wl,-I,`*name* causes the linking phase to override the default name of the dynamic linker, `/usr/lib/ld.so.1`.

The order in which the arguments are passed to a tool with respect to the other specified command line options might change in subsequent compiler releases.

The possible values for *c* are listed in the following table.

TABLE 3-10   `-W` Flags

Flag	Meaning
a	Assembler: (`fbe`); (`gas`)
c	Fortran code generator: (`cg`) *(SPARC)* ;

Flag	Meaning
d	f95 driver
l	Link editor (ld)
m	mcs
O (Capital o)	Interprocedural optimizer
o (Lowercase o)	Postoptimizer
p	Preprocessor (cpp)
0 (Zero)	Compiler (f90comp)
2	Optimizer: (iropt)

Note: You cannot use -Wd to pass f95 options to the Fortran compiler.

3.4.103   —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.

3.4.104   -Xlinker *arg*

Pass *arg* to linker ld(1). Equivalent to —z *arg*

3.4.105   —Xlist[*x*]

(*Solaris only*) 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 3-11   –Xlist Sub-options

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

Option	Feature
–XlistX	Show cross–reference table and errors

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

This option is not available on Linux systems.

3.4.106 –xaddr32[={yes|no}]

(x86/x64 only) The `-xaddr32=yes` compilation flag restricts the resulting executable or shared object to a 32-bit address space.

An executable that is compiled in this manner results in the creation of a process that is restricted to a 32-bit address space. When `-xaddr32=no` is specified a usual 64 bit binary is produced. If the `-xaddr32` option is not specified, `-xaddr32=no` is assumed. If only `-xaddr32` is specified `-xaddr32=yes` is assumed.

This option is only applicable to `-m64` compilations and only on Solaris platforms supporting `SF1_SUNW_ADDR32` software capability. Since Linux kernel does not support address space limitation this option is not available on Linux. The `-xaddr32` option is ignored on Linux.

When linking, if a single object file was compiled with `-xaddr32=yes` the whole output file is assumed to be compiled with `-xaddr32=yes`. A shared object that is restricted to a 32-bit address space must be loaded by a process that executes within a restricted 32-bit mode address space. For more information refer to the `SF1_SUNW_ADDR32` software capabilities definition, described in the *Linker and Libraries Guide*.

3.4.107 –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 3-12   `-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). 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.
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 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*.

3.4.108   –xannotate[={yes|no}]

Create binaries that can later be used by the optimization and observability tools `binopt(1)`, `code-analyzer(1)`, `discover(1)`, `collect(1)`, and `uncover(1)`.

The default on Oracle Solaris is `-xannotate=yes`. The default on Linux is `-xannotate=no`. Specifying `-xannotate` without a value is equivalent to `-xannotate=yes`.

For optimal use of the optimization and observability tools, `-xannotate=yes` must be in effect at both compile and link time. Compile and link with `-xannotate=no` to produce slightly smaller binaries and libraries when optimization and observability tools will not be used.

3.4.109   –xarch=*isa*

Specify instruction set architecture (ISA).

The following table lists the `-xarch` keywords common to both SPARC and x86 platforms.

TABLE 3-13   `-xarch` keywords common to both SPARC and x86 platforms

Flag	Meaning
generic	Uses the instruction set common to most processors. This is the default.
generic64	Compile for good performance on most 64-bit platforms. This option is equivalent to `-m64 -xarch=generic` and is provided for compatibility with earlier releases.
native	Compile for good performance on this system. The compiler chooses the appropriate setting for the current system processor it is running on.
native64	Compile for good performance on 64–bit this system. This option is equivalent to `-m64 -xarch=native` and is provided for compatibility with earlier releases.

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=sparcfmaf ...
```

overrides the `-xarch` 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.

Note the following:

- Object binary files (`.o`) compiled with `generic,sparc, sparcvis2, sparcvis3, sparcfmaf, sparcima` can be linked and can execute together, but can only run on a processor supporting all the instruction sets linked.
- For any particular choice, the generated executable might not run or run much more slowly on legacy architectures. Also, because quad-precision (`REAL*16` and `long double`) floating-point instructions are not implemented in any 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 `generic`.

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

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

`-xarch=`	Meaning (SPARC)
`sparc`	Compile for the SPARC–V9 ISA. Compile for the V9 ISA, but 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 V9 ISA.
`sparc4`	Compile for the SPARC4 version of the SPARC- V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the Ultra SPARC extensions, which includes VIS 1.0, the UltraSPARC-III extensions, which includes VIS2.0, the fused floating-point multiply-add instructions, VIS 3.0, and SPARC4 instructions.
`sparc4b`	Compile for the SPARC4B version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, which includes VIS 1.0, the UltraSPARC-III extensions, which includes VIS2.0, the SPARC64 VI extensions for floating-point multiply-add, the SPARC64 VII extensions for integer multiply-add, and the PAUSE and CBCOND instructions from the SPARC T4 extensions.
`sparc4c`	Compile for the SPARC4C version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, which includes VIS 1.0, the UltraSPARC-III extensions, which includes VIS2.0, the SPARC64 VI extensions for floating-point multiply-add, the SPARC64 VII extensions for integer multiply-add, the VIS3B subset of the VIS 3.0 instructions a subset of the SPARC T3 extensions, called the VIS3B subset of VIS 3.0, and the PAUSE and CBCOND instructions from the SPARC T4 extensions.
`sparc5`	Compile for the SPARC5 version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the extensions, which includes VIS 1.0, the Ultra SPARC-III extensions, which includes VIS2.0, the fused floating-point multiply-add instructions, VIS 3.0, SPARC4, and SPARC5 instructions.
`sparcvis`	Compile for the SPARC–V9 ISA with UltraSPARC extensions. Compile for SPARC-V9 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.

`-xarch=`	Meaning (SPARC)
`sparcvis2`	Compile for the SPARC-V9 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.
`sparcvis3`	Compile for the SPARC VIS version 3 of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the fused multiply-add instructions, and the Visual Instruction Set (VIS) version 3.0
`sparcfmaf`	Compile for the `sparcfmaf` version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, and the SPARC64 VI extensions for floating-point multiply-add. Note that you must use `-xarch=sparcfmaf`in conjunction with `-fma=fused` and some optimization level to get the compiler to attempt to find opportunities to use the multiply-add instructions automatically.
`sparcace`	Compile for the `sparcace` version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the SPARC64 VI extensions for floating-point multiply-add, the SPARC64 VII extensions for integer multiply-add, and the SPARC64 X extensions for ACE floating-point.
`sparcaceplus`	Compile for the `sparcaceplus` version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the SPARC64 VI extensions for floating-point multiply-add, the SPARC64 VII extensions for integer multiply-add, the SPARC64 X extensions for SPARCACE floating-point, and the SPARC64 X+ extensions for SPARCACE floating-point.
`sparcima`	Compile for the `sparcima` version of the SPARC-V9 ISA. Enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0, the SPARC64 VI extensions for floating-point multiply-add, and the SPARC64 VII extensions for integer multiply-add.
`v9`	Equivalent to `-m64 -xarch=sparc` Legacy makefiles and scripts that use `-xarch=v9` to obtain the 64-bit memory model need only use `-m64`.
`v9a`	Equivalent to `-m64 -xarch=sparcvis` and is provided for compatibility with earlier releases.
`v9b`	Equivalent to `-m64 -xarch=sparcvis2` and is provided for compatibility with earlier releases.

Table 3-15 details each of the `-xarch` keywords on x86 platforms. The default on x86 is `generic` (or `generic64` if –m64 is specified) if `-xarch` is not specified.

TABLE 3-15   `-xarch` Values for x86 Platforms

`-xarch=`	Meaning (x86)
`386`	Limits instruction set to the Intel 386/486 architecture.
`pentium_pro`	Limits instruction set to the Pentium Pro architecture.

`-xarch=`	Meaning (x86)
`pentium_proa`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit Pentium Pro architecture.
`sse`	Adds the SSE instruction set to `pentium_pro`. (See Note below.)
`ssea`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE architecture.
`sse2`	Adds the SSE2 instruction set to the `pentium_pro`. (See Note below.)
`sse2a`	Adds the AMD extensions (3DNow!, 3DNow! extensions, and MMX extensions) to the 32-bit SSE2 architecture.
`sse3`	Adds the SSE3 instruction set to the SSE2 instruction set.
`sse3a`	Adds the AMD extended instructions including 3dnow to the SSE3 instruction set.
`amd64`	On Solaris platforms, this is equivalent to `-m64 -xarch=sse2` Legacy makefiles and scripts that use `-xarch=amd64` to obtain the 64-bit memory model should use `-m64`.
`amd64a`	On Solaris platforms, this is equivalent to `—m64 —xarch=sse2a`.
`sse3a`	Adds AMD extended instructions, including 3DNow! to the SSE3 instruction set.
`ssse3`	Adds the SSSE3 instructions to the SSE3 instruction set.
`sse4_1`	Adds the SSE4.1 instructions to the SSSE3 instruction set.
`sse4_2`	Adds the SSE4.2 instructions to the SSE4.1 instruction set.
`amdsse4a`	Adds the SSE4a instructions to the AMD instruction set.
`aes`	Adds the Intel Advanced Encryption Standard instruction set. Note that the compiler does not generate AES instructions automatically when `-xarch=aes` is specified unless the source code includes `.il` inline code, `_asm` statements, or assembler code that use AES instructions, or references to AES intrinsic functions.
`avx`	Uses Intel Advanced Vector Extensions instruction set.
`avx_i`	Uses Intel Advanced Vector Extensions instruction set with the RDRND, FSGSBASE and F16C instruction sets.
`avx2`	Uses Intel Advanced Vector Extensions 2 instruction set.

3.4.109.1   Special Cautions for x86/x64 Platforms:

There are some important considerations when compiling for x86 Solaris platforms.

- If any part of a program is compiled or linked on an x86 platform with –m64, then all parts of the program must be compiled with one of these options as well. For details on the various Intel instruction set architectures (SSE, SSE2, SSE3, SSSE3, and so on) refer to the Intel-64 and IA-32 *Intel Architecture Software Developer's Manual*

- Programs compiled with `-xarch` set to `sse`, `sse2`, `sse2a`, or `sse3` and beyond must be run on platforms supporting these features and extensions..

- With this release, the default instruction set and the meaning of `-xarch=generic` has changed to `sse2`. Now, compiling without specifying a target platform option results in an `sse2` binary incompatible with older Pentium III or earlier systems.

- If you compile and link in separate steps, always link using the compiler and with same `-xarch` setting to ensure that the correct startup routine is linked.

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

- Running programs compiled with these `-xarch` options on platforms that are not enabled with the appropriate features or instruction set extensions could result in segmentation faults or incorrect results occurring without any explicit warning messages.

- This warning extends also to programs that employ `.il` inline assembly language functions or `__asm()` assembler code that utilize SSE, SSE2, SSE2a, and SSE3 (and beyond) instructions and extensions.

3.4.110 **–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 "2.3.1.8 The `ASSUME` Directives" on page 34, 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.

3.4.111 `–xautopar`

Synonym for `-autopar`.

3.4.112 `–xbinopt={prepare | off}`

(SPARC) This option is now obsolete and will be removed in a future release of the compilers. See "3.4.108 –xannotate[={yes|no}]" on page 101

Prepare binary for post-compilation optimization.

The compiled binary file is enabled for later optimizations, transformations, and analysis by binopt(1). This option may be used when building executables or shared objects, and it must be used with an optimization level of `-O1` or higher to be effective.

There will be a modest increase in the size of the binary file when built with this option, on the order of 5%.

If you compile and link in separate steps, `-xbinopt` must appear on both the compile and link steps.

If not all the source code for an application is compiled with `-xbinopt`, the `-xbinopt` flag should still appear on the final link step that builds the program binary, as shown below:

```
example% f95 -0 program -xbinopt=prepare a.o b.o c.f95
```

Only code compiled with `-xbinopt` can be optimized by binopt(1).

Binaries compiled with `-xpg` for `gprof` profiling should not be used with `binopt` as they are incompatible and can result in internal errors.

The default is `-xbinopt=off`.

3.4.113 —xcache=*c*

Define cache properties for the optimizer.

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 3-16   –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[/*t1*]	Define level 1 cache properties.
s1/l1/a1[/*t1*]:*s2/l2/a2*[/*t2*]	Define levels 1 and 2 cache properties.
s1/l1/a1[/*t1*]:*s2/l2/a2*[/*t2*]:*s3/l3/a3*[/*t3*]	Define levels 1, 2, and 3 cache properties

The *si/li/ai/ti* fields 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*
ti	The number of hardware threads sharing the cache at level *i* *(optional)*

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.

3.4.114   **–xcheck[=**_keyword_**[,**_keyword_**]]**

Generate special runtime checks and initializations.

The _keyword_ must be one of the following:

keyword	Feature
stkovf[_action_]	Generate code to detect stack overflow errors at runtime, optionally specifying an action to be taken when a stack overflow error is detected.
	A stack overflow error occurs when a thread's stack pointer is set beyond the thread's allocated stack bounds. The error may not be detected if the new top of stack address is writable.
	A stack overflow error is detected if a memory access violation occurs as a direct result of the error, raising an associated signal (usually SIGSEGV). The signal thus raised is said to be associated with the error.
	If `-xcheck=stkovf[`_action_`]` is specified, the compiler generates code to detect stack overflow errors in cases involving stack frames larger than the system page size. The code includes a library call to force a memory access violation instead of setting the stack pointer to an invalid but potentially mapped address (see `_stack_grow`(3C)).
	The optional _action_, if specified, must be either `:detect` or `:diagnose`.
	If _action_ is `:detect`, a detected stack overflow error is handled by executing the signal handler normally associated with the error.
	If _action_ is `:diagnose`, a detected stack overflow error is handled by catching the associated signal and calling `stack_violation`(3C) to diagnose the error. This is the default behavior if no action is specified.
	If a memory access violation is diagnosed as a stack overflow error, the following message is printed to `stderr`:
	`ERROR: stack overflow detected: pc=<inst_addr>, sp=<sp_addr>`
	where _<inst_addr>_ is the address of the instruction where the error was detected, and _<sp_addr>_ is the value of the stack pointer at the time that the error was detected. After checking for stack overflow and printing the above message if appropriate, control passes to the signal handler normally associated with the error.
	`-xcheck=stkovf:detect` adds a stack bounds check on entry to routines with stack frames larger than system page size (see `_stack_grow`(3C)). The relative cost of the additional bounds check should be negligible in most applications.
	`-xcheck=stkovf:diagnose` adds a system call to thread creation (see `sigaltstack`(2)). The relative cost of the additional system call depends on how frequently the application creates and destroys new threads.
	`-xcheck=stkovf` is supported only on Oracle Solaris. The C runtime library on Linux does not support stack overflow detection.
no%stkovf	Disable runtime checking for stack overflow.
init_local	Perform special initialization of local variables.

keyword	Feature
	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, STATIC and SAVE local variables, and variables in COMMON blocks are not initialized. For details, see the C User's Guide.
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.

3.4.115 −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 3-17   Common −xchip SPARC Processor Names

-xchip=	Optimize for:
generic	most SPARC processors. (This is the default.)
native	this host platform.
sparc64vi	SPARC64 VI processor

`-xchip=`	**Optimize for:**
`sparc64vii`	SPARC64 VII processor
`sparc64viiplus`	SPARC64 VII+ processor
`sparc64x`	SPARC64 X processor
`sparc64xplus`	SPARC64 X+ processor
`ultra`	UltraSPARC processor
`ultra2`	UltraSPARC II processor
`ultra2e`	UltraSPARC IIe processor
`ultra2i`	UltraSPARC IIi processor
`ultra3`	UltraSPARC III processor
`ultra3cu`	UltraSPARC IIIcu processor
`ultra3i`	UltraSPARC IIIi processor
`ultra4`	UltraSPARC IV processor
`ultra4plus`	UltraSPARC IV+ processor
`ultraT1`	UltraSPARC T1 processor
`ultraT2`	UltraSPARC T2 processor
`ultraT2plus`	UltraSPARC T2+ processor
`ultraT3`	UltraSPARC T3 processor
`T3`	SPARC T3 processor
`T4`	SPARC T4 processor
`T5`	Uses the timing properties of the SPARC T5 processor.
`T7`	Uses the timing properties of the SPARC T7 processor.
`M5`	Uses the timing properties of the SPARC M5 processor.
`M6`	Uses the timing properties of the SPARC M6 processor.
`M7`	Uses the timing properties of the SPARC M7 processor.

Note - The following SPARC `-xchip` values are obsolete and may be removed in a future release: `ultra`, `ultra2`, `ultra2e`, `ultra2i`, `ultra3`, `ultra3cu`, `ultra3i`, `ultra4`, and `ultra4plus`.

On x86 platforms, the `-xchip` values are `pentium`, `pentium_pro`, `pentium3`, `pentium4`, `generic`, `opteron`, `core2`, `penryn`, `nehalem`, `amdfam10`, `sandybridge`, `ivybridge`, `haswell`, `westmere`, and `native`.

3.4.116   `-xcode[=`*v*`]`

(SPARC) Specify code address space.

Note - Build shared objects by specifying `-xcode=pic13` or `-xcode=pic32`. While you can build workable shared objects with `-m64 -xcode=abs64` they will be inefficient. Shared objects built with `-m64`, `-xcode=abs32`, or `-m64`, `-xcode=abs44` will not work.

The following table lists the values for *v*.

TABLE 3-18 The `-xcode` Flags

Value	Meaning
abs32	This is the default on 32-bit architectures. Generates 32-bit absolute addresses. Code + data + BSS size is limited to 2**32 bytes.
abs44	This is the default on 64-bit architectures. Generates 44-bit absolute addresses. Code + data + BSS size is limited to 2**44 bytes. Available only on 64–bit architectures.
abs64	Generates 64-bit absolute addresses. Available only on 64-bit architectures.
pic13	Generates position-independent code for use in shared libraries (small model). Equivalent to `-Kpic`. Permits references to at most 2**11 unique external symbols on 32-bit architectures, 2**10 on 64-bit architectures.
pic32	Generates position-independent code for use in shared libraries (large model). Equivalent to `-KPIC`. Permits references to at most 2**30 unique external symbols on 32-bit architectures, 2**29 on 64-bit architectures.

The default is `-xcode=abs32` for 32–bit architectures. The default for 64–bit architectures is `-xcode=abs44`.

When building shared dynamic libraries, the default `-xcode` values of `abs44` and `abs32` will not work with 64–bit architectures. Specify `-xcode=pic13` or `-xcode=pic32` instead. Two nominal performance costs with `-xcode=pic13` and `-xcode=pic32` on SPARC are:.

- A routine compiled with either `-xcode=pic13` or `-xcode=pic32` executes a few extra instructions upon entry to set a register to point at a table (`_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 `_GLOBAL_OFFSET_TABLE_`. If the compilation includes `-xcode=pic32`, there are two additional instructions per global and static memory reference.

When considering these costs, remember that the use of `-xcode=pic13` and `-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 a `.o` file has been compiled with `-xcode=pic13` or `-xcode=pic32` is by using the `nm` command:

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

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

To determine whether to use `-xcode=pic13` or `-xcode=pic32`, check the size of the Global Offset Table (GOT) by using `elfdump -c` looking 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`.See the `elfdump(1)` man page for more information.

Follow these guidelines to determine how you should use `-xcode`:

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

3.4.117 `–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`.

3.4.118 `–xdebugformat={dwarf|stabs}`

Solaris 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=dwarf`.

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=dwarf`. 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.

Use the `dwarfdump(1)` command to determine the format of the debugging information in a compiled object or executable file.

3.4.119   `-xdebuginfo=`*a*`[,`*a*`...]`

Control how much debugging and observability information is emitted.

The term *tagtype* refers to tagged types: `structs`, `unions`, `enums`, and `classes`.

The following list contains the possible values for suboptions *a*. The prefix no% applied to a suboption disables that suboption. The default is `-xdebuginfo=%none`. Specifying `-xdebuginfo` without a suboption is forbidden.

`%none`	No debugging information is generated. This is the default.
`[no%]line`	Emit line number and file information.
`[no%]param`	Emit location list info for parameters. Emit full type information for scalar values (for example, `int`, `char *`) and type names but not full definitions of *tagtypes*.
`[no%]variable`	Emit location list information for lexically global and local variables, including file and function statics but excluding class statics and

	externs. Emit full type information for scalar values such as `int` and `char *` and type names but not full definitions of *tagtypes*.
`[no%]decl`	Emit information for function and variable declarations, member functions, and static data members in `class` declarations.
`[no%]tagtype`	Emit full type definitions of *tagtypes* referenced from `param` and `variable` datasets, as well as template definitions.
`[no%]macro`	Emit macro information.
`[no%]codetag`	Emit DWARF codetags (also known as Stabs `N_PATCH`). This is information regarding bitfields, structure copy, and spills used by RTC and discover.
`[no%]hwcpro`	Generate information critical to hardware counter profiling. This information includes `ldst_map`, a mapping from `ld/st` instructions to the symbol table entry being referenced, and `branch_target` table of branch-target addresses used to verify that backtracking did not cross a branch-target. See `-xhwcprof` for more information.

Note - `ldst_map` requires the presence of *tagtype* information. The driver will issue an error if this requirement is not met.

These are macros which expand to combinations of `-xdebuginfo` and other options as follows:

```
-g = -g2

-gnone =
        -xdebuginfo=%none
        -xglobalize=no
        -xpatchpadding=fix
        -xkeep_unref=no%funcs,no%vars

-g1 =
        -xdebuginfo=line,param,codetag
        -xglobalize=no
        -xpatchpadding=fix
        -xkeep_unref=no%funcs,no%vars

-g2 =
        -xdebuginfo=line,param,decl,variable,tagtype,codetag
        -xglobalize=yes
        -xpatchpadding=fix
        -xkeep_unref=funcs,vars

-g3 =

        -xdebuginfo=line,param,decl,variable,tagtype,codetag,macro
```

```
-xglobalize=yes
-xpatchpadding=fix
-xkeep_unref=funcs,vars
```

3.4.120 **—xdepend**

Synonym for `-depend`.

3.4.121 **—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. (The `f95` `–M`*path* option also passes a regular file to the linker; see the description of the `f95` `–M`*path* option.)

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.

3.4.122 **—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 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 32-bit SPARC file, while big16 describes a 64-bit SPARC 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	big8
64-bit SPARC	big16
32-bit x86	little4
64-bit x86	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.

3.4.122.1 Examples:

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

3.4.122.2 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 `-m64` 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 data types will have different sizes. (However, note that starting with the release of Solaris Studio 12 Update 1, `double:128` is accepted on x64 processors.)

If an option that changes the alignment of the components within a VAX structure (such as `–vax=struct_align`) or a derived type (such as `–aligncommon` or `–dalign`) is used on one platform, the same alignment option has to be used on other platforms sharing the same unformatted data file whose content is affected by the alignment option.

An I/O operation with an 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.

3.4.123 `-xglobalize[={yes|no}]`

Control globalization of file static variables but not functions.

Globalization is a technique needed by fix and continue and interprocedural optimization whereby file static symbols are promoted to global while a prefix is added to the name to keep identically named symbols distinct.

The default is `-xglobalize=no`. Specifying `-xglobalize` is equivalent to specifying `-xglobalize=yes`.

3.4.123.1 Interactions

See `-xpatchpadding`.

-xipo requires globalization as well and will override -xglobalize.

3.4.124 –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.

3.4.125 –xhelp=flags

List the compiler option flags. Equivalent to -help.

3.4.126 —`xhwcprof[={enable | disable}]`

(SPARC) Enable compiler support for dataspace profiling.

With -`xhwcprof` enabled, the compiler generates information that helps tools associate profiled load and store instructions with the data-types and structure members (in conjunction with symbolic information produced with -`g`) to which they refer. It associates profile data with the data space of the target, rather than the instruction space, and provides insight into behavior that is not easily obtained from only instruction profiling.

While you can compile a specified set of object files with -`xhwcprof`, this option is most useful when applied to all object files in the application. This will provide coverage to identify and correlate all memory references distributed in the application's object files.

If you are compiling and linking in separate steps, use -`xhwcprof` at link time as well.

An instance of -`xhwcprof=enable` or -`xhwcprof=disable` overrides all previous instances of -`xhwcprof` in the same command line.

-`xhwcprof` is disabled by default. Specifying -`xhwcprof` without any arguments is the equivalent to -`xhwcprof=enable`.

-`xhwcprof` requires that optimization be turned on and that the debug data format be set to dwarf (-`xdebugformat=dwarf`), which is the default with current Oracle Solaris Studio compilers. The occurrence of -`xhwcprof` and -`xdebugformat=stabs` on the same command line is not permitted.

-`xhwcprof` uses -`xdebuginfo` to automatically enable the minimum amount of debugging information it needs, so -`g` is not required.

The combination of -`xhwcprof` and -`g` increases compiler temporary file storage requirements by more than the sum of the increases due to -`xhwcprof` and -`g` specified alone.

-`xhwcprof` is implemented as a macro that expands to various other, more primitive, options as follows:

```
-xhwcprof
       -xdebuginfo=hwcprof,tagtype,line
-xhwcprof=enable
       -xdebuginfo=hwcprof,tagtype,line
-xhwcprof=disable
       -xdebuginfo=no%hwcprof,no%tagtype,no%line
```

The following command compiles `example.f` and specifies support for hardware counter profiling and symbolic analysis of data types and structure members using DWARF symbols:

```
f95 -c -O -xhwcprof -g  example.f
```

For more information on hardware counter-based profiling, see the *Oracle Solaris Studio Performance Analyzer* manual.

3.4.127 `–xia[={widestneed|strict}]`

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

The default if not specified is `-xia=widestneed`.

Fortran extensions for interval arithmetic calculations are detailed in the *Interval Arithmetic Programming Reference.* See also "3.4.132 – `xinterval[={widestneed|strict|no}]`" on page 123.

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`

3.4.128 `–xinline=`*list*

Synonym for `-inline`.

3.4.129 `-xinline_param=`*a*`[,`*a*`[,`*a*`]...]`

Use this option to manually change the heuristics used by the compiler for deciding when to inline a function call.

This option only has an effect at -O3 or higher. The following sub-options have an effect only at -O4 or higher when automatic inlining is on.

In the following sub-options *n* must be a positive integer; *a* can be one of the following:

TABLE 3-19   `-xinline_param` Sub-options

Sub-option	Meaning
`default`	Set the values of all the sub-options to their default values.
`max_inst_hard[:`*n*`]`	Automatic inlining only considers functions smaller than n pseudo instructions (counted in compiler's internal representation) as possible inline candidates.

Sub-option	Meaning
	Under no circumstances will a function larger than this be considered for inlining.
max_inst_soft[:*n*]	Set inlined function's size limit to *n* pseudo instructions (counted in compiler's internal representation). Functions of greater size than this may sometimes be inlined. When interacting with max_inst_hard, the value of max_inst_soft should be equal to or smaller than the value of max_inst_hard, i.e, max_inst_soft <= max_inst_hard. In general, the compiler's automatic inliner only inlines calls whose called function's size is smaller than the value of max_inst_soft. In some cases a function may be inlined when its size is larger than the value of max_inst_soft but smaller than that of max_inst_hard. An example of this would be if the parameters passed into a function were constants. When deciding whether to change the value of max_inst_hard or max_inst_soft for inlining one specific call site to a function, use -xinline_report=2 to report detailed inlining message and follow the suggestion in the inlining message.
max_function_inst[:*n*]	Allow functions to increase due to automatic inlining by up to *n* pseudo instructions (counted in compiler's internal representation).
max_growth[:*n*]	The automatic inliner is allowed to increase the size of the program by up to *n*% where the size is measured in pseudo instructions.
min_counter[:*n*]	The minimum call site frequency counter as measured by profiling feedback (-xprofile) in order to consider a function for automatic inlining. This option is valid only when the application is compiled with profiling feedback (-xprofile=use).

level[:*n*]	Use this suboption to control the degree of automatic inlining that is applied. The compiler will inline more functions with higher settings for -xinline_param=level.
	
	n must be one of 1, 2, or 3.
	
	The default value of *n* is 2 when this option is not specified, or when the options is specified without :*n*.
	
	Specify the level of automatic inline:
	
	`level:1` basic inlining
	`level:2` medium inlining (default)
	`level:3` aggressive inlining
	
	The level decides the specified values for the combination of the following inlining parameters:
	
	`max_growth`
	`+ max_function_inst`
	`+ max_inst`
	`+ max_inst_call`
	
	When level = 1, all the parameters are half the values of the default.

Sub-option	Meaning
	When `level` = 2, all the parameters are the default value.
	When `level` = 3, all the parameters are double the values of the default.
`max_recursive_depth[:`n`]`	When a function calls itself either directly or indirectly, it is said to be making a recursive call.
	This suboption allows a recursive call to be automatically inlined up to n levels.
`max_recursive_inst[:`n`]`	Specifies the maximum number of pseudo instructions (counted in compiler's internal representation) the caller of a recursive function can grow to by performing automatic recursive inlining.
	When interactions between `max_recursive_inst` and `max_recursive_depth` occur, recursive function calls will be inlined until either the `max_recursive_depth` number of recursive calls, or until the size of the function being inlined into exceeds `max_recursive_inst`. The settings of these two parameters control the degree of inlining of small recursive functions.

If `-xinline_param=default` is specified, the compiler will set all the values of the subopitons to the default values.

If the option is not specified, the default is `-xinline_param=default`.

The list of values and options accumulate from left to right. So for a specification of `-xinline_param=max_inst_hard:30,..,max_inst_hard:50`, the value `max_inst_hard:50` will be passed to the compiler.

If multiple `-xinline_param` options are specified on the command line, the list of sub-options likewise accumulate from left to right. For example, the effect of

```
 -xinline_param=max_inst_hard:50,min_counter:70 ...
   -xinline_param=max_growth:100,max_inst_hard:100
```

will be the same as that of

```
-xinline_param=max_inst_hard:100,min_counter:70,max_growth:100
```

3.4.130 `-xinline_report[=`n`]`

This option generates a report written to standard output on the inlining of functions by the compiler. The type of report depends on the value of n, which must be 0, 1, or 2.

0   No report is generated.

1   A summary report of default values of inlining parameters is generated.

| 2 | A detailed report of inlining messages is generated, showing which callsites are inlined and which are not, with a short reason for not inlining a callsite. In some cases, this report will include suggested values for `-xinline_param` that can be used to inline a callsite that is not inlined. |

When `-xinline_report` is not specified, the default value for *n* is 0. When `-xinline_report` is specified without =*n*, the default value is 1.

When `-xlinkopt` is present, the inlining messages about the callsites that are not inlined might not be accurate.

The report is limited to inlining performed by the compiler that is subject to the heuristics controllable by the `-xinline_param` option. Callsites inlined by the compiler for other reasons may not be reported.

3.4.131 `–xinstrument=[%no]datarace`

Specify this option to compile and instrument your program for analysis by the Thread Analyzer.

(For information on the Thread Analyzer, see `tha`(1) for details.)

By compiling with this option you can then use the Performance Analyzer to run the instrumented program with `collect -r races` to create a data-race-detection experiment. You can run the instrumented code standalone but it runs more slowly.

Specify `-xinstrument=no%datarace` to turn off this feature. This is the default.

`-xinstrument` must be specified with an argument.

If you compile and link in separate steps, you must specify `-xinstrument=datarace` in both the compilation and linking steps.

This option defines the preprocessor token `__THA_NOTIFY`. You can specify `#ifdef __THA_NOTIFY` to guard calls to `libtha`(3) routines.

This option also sets `-g`.

3.4.132 `–xinterval[={widestneed|strict|no}]`

(Oracle Solaris) 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 extensions for interval arithmetic calculations are detailed in the *Fortran 95 Interval Arithmetic Programming Reference.* See also "3.4.127 – xia[={widestneed|strict}]" on page 120.

3.4.133 `−xipo[={0|1|2}]`

Perform interprocedural optimizations.

Performs whole-program optimizations by invoking an interprocedural analysis pass. `-xipo` performs 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
- Building executables compiled with -xipo using a parallel make tool can cause problems if object files used in the build are common to the link steps running in parallel. Each link step should have its own copy of the object file being optimized prior to linking.
- 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
- If you have .o files compiled with the -xipo option from different compiler versions, mixing these files can result in failure with an error message about "IR version mismatch". When using the -xipo option, all the files should be compiled with the same version of the compiler.
- 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`.

3.4.134 `—xipo_archive[={none|readonly|writeback}]`

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

The value must be one of the following:

`none`	No processing of archive files is performed. The compiler does not apply cross-module inlining or other cross-module optimizations to object files compiled using `-xipo` and extracted from an archive library at link time. To do that, both `-xipo` and either `-xipo_archive=readonly`or `-xipo_archive=writeback` must be specified at link time.
`readonly`	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the archive library (`.a`) before producing an executable.
	The option `-xipo_archive=readonly` enables cross-module inlining and interprocedural data flow analysis of object files in an archive library specified at link time. However, it does not enable cross-module optimization of the archive library's code except for code that has been inserted into other modules by cross module inlining.
	To apply cross-module optimization to code within an archive library, `-xipo_archive=writeback` is required. Note that doing so modifies the contents of the archive library from which the code was extracted.

`writeback`	The compiler optimizes object files passed to the linker with object files compiled with `-xipo` that reside in the archive library (`.a`) before producing an executable. Any object filed contained in the library that were optimized during the compilation are replaced with their optimized version.
	For parallel links that use a common set of archive libraries, each link should create its own copy of archive libraries to be optimized before linking.

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

3.4.135 `-xipo_build=[yes|no]`

Building `-xipo` without `-xipo_build` involves two passes through the compiler—once when producing the object files, and then again later at link time when performing the cross file optimization. Setting `-xipo_build` reduces compile time by avoiding optimizations during the initial pass and optimizing only at link time. Optimization is not needed for the object files, as with `-xipo` it will be performed at link time. If unoptimized object files built with `-xipo_build` are linked without including `-xipo` to perform optimization, the application will fail to link with an unresolved symbol error.

3.4.135.1 `-xipo_build` Examples

The following example performs a fast build of `.o` files, followed by crossfile optimization at link time:

```
% cc -O -xipo -xipo_build -o code1.o -c code1.c
% cc -O -xipo -xipo_build -o code2.o -c code2.c
% cc -O -xipo -o a.out code1.o code2.o
```

The `-xipo_build` will turn off `-O` when creating the `.o` files, to build these quickly. Full `-O` optimization will be performed at link time as part of `-xipo` crossfile optimization.

The following example links without using `-xipo`.

```
% cc -O -o a.out code1.o code2.o
```

If either `code1.o` or `code2.o` were generated with `-xipo_build`, the result will be a link-time failure indicating the symbol `__unoptimized_object_file` is unresolved.

When building `.o` files separately, the default behavior is `-xipo_build=no`. However, when the executable or library is built in a single pass from source files, `-xipo_build` will be implicitly enabled. For example:

```
% cc -fast -xipo a.c b.c c.c
```

will implicitly enable `-xipo_build=yes` for the first passes that generate `a.o`, `b.o`, and `c.o`. Include the option `-xipo_build=no` to disable this behavior.

3.4.136   `–xivdep[=`*p*`]`

Disable or set interpretation of `!DIR$ IVDEP` directives

The `IVDEP` directive tells the compiler to ignore some or all loop-carried dependences on array references that it finds in a loop, allowing it to perform various loop optimizations such as microvectorization, distribution, software pipelining, among others, that would not be otherwise possible. It is employed in situations where the user knows either that the dependences do not matter or that they never occur in practice.

The interpretation of `!DIR$ IVDEP` directives depend upon the value of the `-xivdep` option. The following values for *p* are interpreted as follows:

`loop` — ignore assumed loop-carried vector dependences
`loop_any` — ignore all loop-carried vector dependences
`back` — ignore assumed backward loop-carried vector dependences
`back_any` — ignore all backward loop-carried vector dependences
`none` — do not ignore any dependences (disables `IVDEP` directives)

These interpretations are provided for compatibility with other vendor's interpretations of the `IVDEP` directive.

The default when `–xivdep` is not specified, and when `–xivdep` is specified without an argument, are both `–xivdep=loop`, which implies that the `!DIR$ IVDEP` directives are enabled by default.

For more information, see "2.3.3 `IVDEP` Directive" on page 36 for details.

3.4.137   `-xjobs{=`*n*`|auto}`

Compile with multiple processes. If this flag is not specified, the default behavior is `-xjobs=auto`.

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. Currently, `-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.

When `-xjobs=auto` is specified, the compiler will automatically choose the appropriate number of parallel jobs.

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

If `-xjobs` is not specified, the default behavior is `-xjobs=auto`. This can be overridden by adding `-xjobs=`*n* to the command line. Multiple instances of `-xjobs` on the command line override each other until the right-most instance is reached.

3.4.137.1 `-xjobs` Examples

The following example links with up to three parallel processes for `-xipo`:

```
% cc -xipo -xO4 -xjobs=3 t1.o t2.o t3.o
```

The following example links serially with a single process for `-xipo`:

```
% cc -xipo -xO4 -xjobs=1 t1.o t2.o t3.o
```

The following example links in parallel, with the compiler choosing the number of jobs for `-xipo`:

```
% cc -xipo -xO4  t1.o t2.o t3.o
```

Note that this is exactly the same behavior as when explicitly specifying `-xjobs=auto`:

```
% cc -xipo -xO4 -xjobs=auto t1.o t2.o t3.o
```

3.4.138 `-xkeep_unref[={[no%]funcs,[no%]vars}]`

Keep definitions of unreferenced functions and variables. The `no%` prefix allows the compiler to potentially remove the definitions.

The default is `no%funcs,no%vars`. Specifying `-xkeep_unref` is equivalent to specifying `-xkeep_unref=funcs,vars`, meaning that `-keep_unref` keeps everything.

3.4.139 `–xkeepframe[=[%all,%none,`*name*`,no%`*name*`]]`

Prohibit stack related optimizations for the named functions (*name*).

| %all | Prohibit stack related optimizations for all the code. |
| %none | Allow stack related optimizations for all the code. |

This option is accumulative and can appear multiple times on the command line. For example, –xkeepframe=%all –xkeepframe=no%func1 indicates that the stack frame should be kept for all functions except func1. Also, –xkeepframe overrides –xregs=frameptr. For example, –xkeepframe=%all –xregs=frameptr indicates that the stack should be kept for all functions, but the optimizations for –xregs=frameptr would be ignored.

If not specified on the command line, the compiler assumes -xkeepframe=%none as the default. If specified but without a value, the compiler assumes -xkeepframe=%all

3.4.140   **–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 use the BLAS routines in the Sun Performance Library or inline the routines.

The –library=sunperf option is needed to link with 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

`-xknown_lib=`	**Feature**
`blas`	Selects all the BLAS routines. Equivalent to `-xknown_lib=blas1,blas2,blas3`
`intrinsics`	The compiler ignores any explicit `EXTERNAL` declarations for Fortran intrinsics, thereby ignoring any user-supplied intrinsic routines. (See the *Fortran Library Reference* for lists of intrinsic function names.)

3.4.141 `–xl`

(Obsolete) This legacy `f77` option is no longer supported. For the equivalent options in the current Fortran compiler, use: `-f77=%all,no%backslash -vax=$all,no%debug`

3.4.142 `–xlang=f77`

(SPARC) Prepare for linking with runtime libraries compatible with objects created by earlier versions of the legacy `f77` compiler.

`f95 -xlang=f77` implies linking with the `f77compat` library, and is a shorthand way for linking `f95` 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`.

To determine which driver to use for mixed-language linking, use the following language hierarchy:

C++	Use the `CC` command. See the *C++ Uder's Guide* for details.
Fortran 95 (or Fortran 90)	Use the `f95` command.
Fortran 77	Use `f95 -xlang=f77`.

C	Use the `cc` command. See the *C Uder's Guide* for details.

3.4.143 `–xld`

(Obsolete) This (`f77`) option is no longer supported. For the equivalent options in the current Fortran compiler, use: `-f77=%all,no%backslash -vax=$all,no%debug`

3.4.144 `–xlibmil`

Synonym for `-libmil`.

3.4.145 `–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.

3.4.146 `–xlic_lib=sunperf`

Obsolete. Use `–library=sunperf` to link with the Sun Performance Library.

3.4.147 `–xlinkopt[={1|2|0}]`

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 post-optimization 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 with the -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 optimizes code coming from static libraries that appear on the compiler command line, but does 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 -xlinkopt option requires profile feedback (-xprofile) in order to optimize the program. Profiling reveals the most- and least-used parts of the code, which enables the optimizer to focus its effort accordingly. Link-time optimization is particularly important with large applications where optimal placement of code can substantially reduce instruction cache misses. Additionally, -xlinkopt is most effective when used to compile the whole program. Use this option as follows:

```
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 executable by including debugging information.

3.4.148 `—xloopinfo`

Synonym for `-loopinfo`.

3.4.149 `—xM`

Generate make dependencies.

This option produces make dependencies for the compiled source file on standard output. The option covers all make dependencies for the source file, both header files and Fortran modules.

For module dependencies, this option uses an object-based module dependency scheme to eliminate the need for explicit build rules to create the module files.

This compilation cannot be used with `-c`, `-S`, `-Xlist`, or any other compilation options that produce different compilation outputs.

The generated dependency output does not contain any build rules, only dependencies for the files. The user will need to specify the build rules for all the files needed for the build. However, for the module files, no explicit build rules are needed, as the module files are created at the same time as the associated object files. Therefore, the module files only need to have a generic build rule:

```
%.mod:
        @ echo $@ is already up to date.
```

The module file build rule is only needed to prevent the 'make' process from stripping all the dependencies related to module files if there are no build rules for them. Other than that, the build rule does not do anything, as shown in the example above.

When used with the `-keepmod` option, the dependencies generated by the `-xM` option will prevent compilation cascade due to the unnecessarily updated modules files, as well as prevent the problem with recompilation on the same source files due to the use of the `-keepmod` option to prevent unnecessary updates on the module files.

This option works in conjunction with the `-M`, `-I`, and `-moddir` options to determine the appropriate directories for the module files needed in the build. Pre-compiled module files, for example those shipped by third parties, should be located at a directory pointed to by the `-M` option so the correct dependencies can be generated.

3.4.150 **–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 `!$PRAGMA SUN OPT=`_n_ directive when it appears in the source input. Without this option, the compiler treats these lines as comments. See .

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`.

3.4.151 **–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   On 64–bit platforms, raise signal SIGBUS only for alignments less or
                               equal to 4, otherwise interpret access and continue execution. On other
                               platforms f is equivalent to i.

The defaults when compiling without -xmemalign specified are:

- 8i for 32–bit platforms
- 8s for 64–bit platforms with C and C++
- 8f for 64–bit platforms 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.

You must also specify -xmemalign whenever you link to an object file that was compiled with a b value of either i or f.

The -dalign option is a macro:

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

Do not use -aligncommon=1 with -xmemalign as these declarations will conflict and could cause a segmentation fault on some platforms and configurations.

See "3.4.1 –aligncommon[={1|2|4|8|16}]" on page 51 for details.

3.4.152 —xmodel=[small | kernel | medium]

(x86) Specify the data address model for shared objects on Solaris x64 platforms.

The -xmodel option enables the compiler to create 64- bit shared objects for the Oracle Solaris x64 platforms and should only be specified for the compilation of such objects.

This option is valid only when -m64 is specified on 64–bit enabled x86 platforms ("x64").

small   This option generates code for the small model in which the virtual
                               address of code executed is known at link time and all symbols are

known to be located in the virtual addresses in the range from 0 to 2^{31} – 2^{24} - 1.

kernel   Generates code for the kernel model in which all symbols are defined to be in the range from 2^{64} - 2^{31} to 2^{64} - 2^{24}.

medium   Generates code for the medium model in which no assumptions are made about the range of symbolic references to data sections. Size and address of the text section have the same limits as the small code model. Applications with large amounts of static data might require -xmodel=medium when compiling with –m64.

If you do not specify -xmodel, the compiler assumes -xmodel=small. Specifying -xmodel without an argument is an error.

It is not necessary to compile all routines with this option as long as you an ensure that objects being accessed are within range.

3.4.153   **–xnolib**

Synonym for -nolib.

3.4.154   **–xnolibmil**

Synonym for -nolibmil.

3.4.155   **–xnolibmopt**

Do not use fast math library.

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

f95 -fast -xnolibmopt …

3.4.156   **–xO***n*

Synonym for -O*n*.

3.4.157   `-xopenmp[={parallel|noopt|none}]`

Enable explicit parallelization with OpenMP directives.

The flag accepts the following suboption keywords:

`parallel`   Enables recognition of OpenMP pragmas. The optimization level under `-xopenmp=parallel` is `-xO3`. The compiler raises the optimization level to `-xO3` if necessary and issues a warning.

This flag also defines the preprocessor macro `_OPENMP`. The `_OPENMP` macro is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports. Refer to the *Oracle Solaris Studio OpenMP API User's Guide* for the value of the `_OPENMP` macro for a particular release.

`noopt`   Enables recognition of OpenMP pragmas. The compiler does not raise the optimization level if it is lower than `-xO3`. If you explicitly set the optimization lower than `-xO3`, as in `f95 -xO2 -xopenmp=noopt`, the compiler issues an error. If you do not specify an optimization level with `-xopenmp=noopt`, the OpenMP pragmas are recognized, the program is parallelized accordingly, but no optimization is done. This suboption also defines the preprocessor macro `_OPENMP`.

`none`   Does not enable the recognition of OpenMP pragmas, makes no change to the optimization level of your program, and does not define any preprocessor macros. This is the default when `-xopenmp` is not specified.

If you specify `-xopenmp`, but do not specify a suboption keyword, the compiler assumes `-xopenmp=parallel`. If you do not specify `-xopenmp` at all, the compiler assumes `-xopenmp=none`.

Suboptions `parallel` and `noopt` will invoke `-stackvar` automatically.

If you are debugging an OpenMP program with `dbx`, compile with `-g -xopenmp=noopt` so you can breakpoint within parallel regions and display the contents of variables.

The default for `-xopenmp` might change in a future release. You can avoid warning messages by explicitly specifying an appropriate optimization level.

Use the `OMP_NUM_THREADS` environment variable to specify the number of threads to use when running an OpenMP program. If `OMP_NUM_THREADS` is not set, the default number of threads used to execute a parallel region is the number of cores available on the machine, capped at 32. You can specify a different number of threads by setting the `OMP_NUM_THREADS` environment

variable, or by calling the `omp_set_num_threads()` OpenMP runtime routine, or by using the `num_threads` clause on the parallel region directive. For best performance, the number of threads used to execute a parallel region should not exceed the number of hardware threads (or virtual processors) available on the machine. On Oracle Solaris systems, this number can be determined by using the `psrinfo(1M)` command. On Linux systems, this number can be determined by inspecting the file `/proc/cpuinfo`. See the *OpenMP API User's Guide* for more information.

Nested parallelism is disabled by default. To enable nested parallelism, you must set the `OMP_NESTED` environment variable to `TRUE`. See the *OpenMP API User's Guide*.

If you compile and link in separate steps, specify `-xopenmp` in both the compilation step and the link step. When used with the link step, the `-xopenmp` option will link with the OpenMP runtime support library, `libmtsk.so`.

For up-to-date functionality and performance, make sure that the latest patch of the OpenMP runtime library, `libmtsk.so`, is installed on the system.

For more information about the OpenMP Fortran 95, C, and C++ application program interface (API) for building multithreaded applications, see the *Oracle Solaris Studio OpenMP API User's Guide*.

3.4.158 **–xpad**

Synonym for `-pad`.

3.4.159 **–xpagesize=***size*

Set the preferred page size for the stack and the heap.

On SPARC platforms, the *size* value must be one of the following:

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

On x86 platforms, the *size* value must be one of the following:

`4K 2M 4M` or `default`

For example: `-xpagesize=4M`

Not all these page sizes are supported on all platforms and depend on the architecture and Oracle Solaris environment. The page size specified must be a valid page size for the Oracle Solaris operating environment on the target platform. If it is not, the request will be silently ignored at run-time.

Use the pagesize(1) Oracle Solaris command to determine the number of bytes in a page. The operating system offers no guarantee that the page size request will be honored. However, appropriate segment alignment can be used to increase the likelihood of obtaining the requested page size. See the -xsegment_align option on how to set the segment alignment. You can use pmap(1) or meminfo(2) to determine page size of the target platform.

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 the combination -xpagesize_heap=*size* -xpagesize_stack=*size*. These two options accept the same arguments as -xpagesize. 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 Oracle Solaris command ppgsz(1) with the equivalent options, before starting the program. See the Oracle Solaris man pages for details.

3.4.160   **–xpagesize_heap=***size*

Set the preferred page size for the heap.

The *size* value is the same as described for -xpagesize.

See -xpagesize for details.

3.4.161   **–xpagesize_stack=***size*

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

The *size* value is the same as described for -xpagesize.

See -xpagesize for details.

3.4.162   **-xpatchpadding[={fix|patch|***size***}]**

Reserve an area of memory before the start of each function. If fix is specified, the compiler will reserve the amount of space required by fix and continue. This is the first default. If either

`patch` or no value is specified, the compiler will reserve a platform-specific default value. A value of `-xpatchpadding=0` will reserve 0 bytes of space. The maximum value for *size* on x86 is 127 bytes and on SPARC is 2048 bytes.

3.4.163   **−xpec[={yes|no}]**

Generate a PEC (Portable Executable Code) binary.

This option puts the program intermediate representations in the object file and the binary. This binary may be used later for tuning and troubleshooting.

A binary built with `-xpec` is usually 5 to 10 times larger than if it is built without. The default is `-xpec=no`.

Without an argument, `-xpec` is equivalent to `-xpec=yes`.

3.4.164   **−xpg**

Synonym for `-pg`.

3.4.165   **−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`.

3.4.166   **−xprefetch[=*a*[,*a*]]**

Enable prefetch instructions on those architectures that support prefetch.

See "2.3.1.7 The PREFETCH Directives" on page 33 for a description of the Fortran PREFETCH directives.

a must be one of the following:

`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*	(SPARC) 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.

> If you are running computationally intensive codes on large SPARC 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.

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.

3.4.166.1   Defaults:

If -xprefetch is not specified, -xprefetch=auto,explicit is assumed.

If only -xprefetch is specified, -xprefetch=auto,explicit is assumed.

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.

3.4.166.2   Interactions:

With -xprefetch=explicit, the compiler will recognize the directives:

```
!$PRAGMA SUN_PREFETCH_READ_ONCE (name)
!$PRAGMA SUN_PREFETCH_READ_MANY (name)
!$PRAGMA SUN_PREFETCH_WRITE_ONCE (name)
!$PRAGMA SUN_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 (auto) is enabled on SPARC processors.

3.4.166.3   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.

3.4.167 —xprefetch_auto_type=indirect_array_access

Generate indirect prefetches for a data arrays accessed indirectly.

Generates 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%` can be added to negate the declaration.

The default is `-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.

3.4.168 —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.

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 may note be effective on older SPARC or x86 platforms.

3.4.169 —xprofile=*p*

Collects data for a profile or uses a profile to optimize.

p must be `collect`[:*profdir*], `use`[:*profdir*], or `tcov`[:*profdir*].

This option causes execution frequency data to be collected and saved during execution, then the data can be used in subsequent runs to improve performance. Profile collection is safe for multithreaded applications. That is, profiling a program that does its own multitasking (`-mt`) produces accurate results. This option is only valid when you specify `-xO2` or greater level of

optimization. If compilation and linking are performed in separate steps, the same -xprofile option must appear on the link step as well as the compile step.

collect[:*profdir*]   Collects and saves execution frequency for later use by the optimizer with -xprofile=use. The compiler generates code to measure statement execution-frequency.

-xMerge, -ztext, and -xprofile=collect should not be used together. While -xMerge forces statically initialized data into read-only storage, -ztext prohibits position-dependent symbol relocations in read-only storage, and -xprofile=collect generates statically initialized, position-dependent symbol relocations in writable storage.

The profile directory name *profdir*, if specified, is the pathname of the directory where profile data are to be stored when a program or shared library containing the profiled object code is executed. If the *profdir* pathname is not absolute, it is interpreted relative to the current working directory when the program is compiled with the option -xprofile=use:*profdir*.

If no profile directory name is specified with –xprofile=collect:*prof_dir* or –xprofile=tcov:*prof_dir*, profile data are stored at run time in a directory named *program*.profile where *program* is the basename of the profiled process's main program. In this case, the environment variables SUN_PROFDATA and SUN_PROFDATA_DIR can be used to control where the profile data are stored at run time. If set, the profile data are written to the directory given by $SUN_PROFDATA_DIR/$SUN_PROFDATA. If a profile directory name is specified at compilation time, SUN_PROFDATA_DIR and SUN_PROFDATA have no effect at run time. 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.

If these environment variables are not set, the profile data is written to the directory *profdir*.profile in the current directory, where *profdir* is the name of the executable or the name specified in the -xprofile=collect:*profdir* flag. -xprofile does not append. profile to *profdir* if *profdir* already ends in .profile. If you run the program several times, the execution frequency data accumulates in the *profdir*.profile directory; that is output from prior executions is not lost.

If you are compiling and linking in separate steps, make sure that any object files compiled with -xprofile=collect are also linked with -xprofile=collect.

use[:*profdir*]   Uses execution frequency data collected from code compiled with –xprofile=collect[:*profdir*] or –xprofile=tcov[:*profdir*] to optimize

for the work performed when the profiled code was executed. *profdir* is the pathname of a directory containing profile data collected by running a program that was compiled with –xprofile=collect[:*profdir*] or –xprofile=tcov[:*profdir*].

To generate data that can be used by both tcov and –xprofile=use[:*profdir*], a profile directory must be specified at compilation time, using the option –xprofile=tcov[:*profdir*]. The same profile directory must be specified in both –xprofile=tcov:*profdir* and –xprofile=use:*profdir*. To minimize confusion, specify *profdir* as an absolute pathname.

The *profdir* pathname is optional. If *profdir* is not specified, the name of the executable binary is used. a.out is used if -o is not specified. The compiler looks for *profdir*.profile/feedback, or a.out.profile/feedback when *profdir* is not specified. For example:

```
demo% f95 -xprofile=collect -o myexe prog.f95
demo% f95 -xprofile=use:myexe -xO5 -o myexe prog.f95
```

The program is optimized by using the execution frequency data previously generated and saved in the feedback files written by a previous execution of the program compiled with -xprofile=collect.

Except for the -xprofile option, the source files and other compiler options must be exactly the same as those used for the compilation that created the compiled program that generated the feedback file. The same version of the compiler must be used for both the collect build and the use build as well.

If compiled with -xprofile=collect:*profdir*, the same profile directory name *profdir* must be used in the optimizing compilation: -xprofile=use:*profdir*.

See also -xprofile_ircache for speeding up compilation between collect and use phases.

tcov[:*profdir*]   Instrument object files for basic block coverage analysis using tcov(1).

If the optional *profdir* argument is specified, the compiler will create a profile directory at the specified location The data stored in the profile directory can be used either by tcov(1) or by the compiler with -xprofile=use:*profdir*. If the optional *profdir* pathname is omitted, a profile directory will be created when the profiled program is executed. The data stored in the profile directory can only be used by tcov(1). The location of the profile directory can be controlled using environment variables SUN_PROFDATA and SUN_PROFDATA_DIR.

If the location specified by *profdir* is not an absolute pathname, it is interpreted at compilation time relative to the current working directory

at the time of compilation. If *profdir* is specified for any object file, the same location must be specified for all object files in the same program. The directory whose location is specified by *profdir* must be accessible from all machines where the profiled program is to be executed. The profile directory should not be deleted until its contents are no longer needed, because data stored there by the compiler cannot be restored except by recompilation.

Example [1]: if object files for one or more programs are compiled with `-xprofile=tcov:/test/profdata`, a directory named `/test/profdata.profile` will be created by the compiler and used to store data describing the profiled object files. The same directory will also be used at execution time to store execution data associated with the profiled object files.

Example [2]: if a program named `myprog` is compiled with `-xprofile=tcov` and executed in the directory `/home/joe`, the directory `/home/joe/myprog.profile` will be created at run time and used to store runtime profile data.

3.4.170   **–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.

3.4.171   **–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.

3.4.172 `—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.

3.4.173 `—xreduction`

Synonym for `-reduction`.

3.4.174 **–xregs=***r*

Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following suboptions: `appl`, `float`,`frameptr`.

Prefixing a suboption with `no%` disables that suboption.

Note that –`xregs` suboptions are restricted to specific hardware platforms.

Example: `-xregs=appl,no%float`

TABLE 3-20   The `-xregs` Suboptions

Value	Meaning
`appl`	(SPARC) Allow the compiler to generate code using the application registers as scratch registers. The application registers are:
	`g2`, `g3`, `g4` (on 32–bit platforms)
	`g2`, `g3` (on 64–bit platforms)
	It is strongly recommended that all system software and libraries be compiled using `-xregs=no%appl`. System software (including shared libraries) must preserve these registers' values for the application. Their use is intended to be controlled by the compilation system and must be consistent throughout the application.
	In the SPARC ABI, these registers are described as *application* registers. Using these registers can improve performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
`float`	(SPARC) Allow the compiler to generate code by using the floating-point registers as scratch registers for integer values. Use of floating-point values may use these registers regardless of this option. If you want your code to be free of all references to floating point registers, you need to use `-xregs=no%float` and also make sure your code does not in any way use floating point types.
`frameptr`	(x86) Allow the compiler to use the frame-pointer register (`%ebp` on IA32, `%rbp` on AMD64) as a general-purpose register.
	The default is `-xregs=no%frameptr`
	With `-xregs=framptr` the compiler is free to use the frame-pointer register to improve program performance. However, some features of the debugger and performance measurement tools may be limited as a result. Stack tracing, debuggers, and performance anayzers cannot report on functions compiled with –`xregs=frameptr`
	Mixed C, Fortran, and C++ code should not be compiled with –`xregs=frameptr` if a C++ function, called directly or indirectly from a C or Fortran function, can throw an exception. If compiling such mixed source code with –`fast`, add –`xregs=no%frameptr` after the –`fast` option on the command line.

Value	Meaning
	With more available registers on 64–bit platforms, compiling with –xregs=frameptr has a better chance of im proving 32–bit code performance than 64–bit code.
	The compiler ignores -xregs=frameptr and issues a warning if you also specify -pg. Also, -xkeepframe overrides -xregs=frameptr.

The SPARC default is -xregs=appl,float.

The x86 default is -xregs=no%frameptr. -xregs=frameptr in included in the expansion of -fast.

It is strongly recommended that you compile code intended for shared libraries that will link with applications, with -xregs=no%appl,float. At the very least, the shared library should explicitly document how it uses the application registers so that applications linking with those libraries are aware of these register assignments.

For example, an application using the registers in some global sense (such as using a register to point to some critical data structure) would need to know exactly how a library with code compiled without -xregs=no%appl is using the application registers in order to safely link with that library.

On x86 systems, -pg is incompatible with -xregs=frameptr, and these two options should not be used together. Note also that -xregs=frameptr is included in -fast.

3.4.175 `-xs[={yes|no}]`

(Oracle Solaris) Link debug information from object files into executable.

-xs is the same as -xs=yes.

The default for -xdebugformat=dwarf is the same as -xs=yes.

The default for -xdebugformat=stabs is the same as -xs=no.

This option controls the trade-off of executable size versus the need to retain object files in order to debug. For dwarf, use -xs=no to keep the executable small but depend on the object files. For stabs, use -xs or -xs=yes to avoid dependence on the object files at the cost of a larger executable. This option has almost no effect on dbx performance or the runtime per formance of the program.

When the compile command forces linking (that is, -c is not specified) there will be no object file(s) and the debug information must be placed in the executable. In this case, -xs=no (implicit or explicit) will be ignored.

The feature is implemented by having the compiler adjust the section flags and/or section names in the object file that it emits, which then tells the linker what to do for that object file's debug information. It is therefore a compiler option, not a linker option. It is possible to have an executable with some object files compiled `-xs=yes` and others compiled `-xs=no`.

Linux compilers accept but ignore `-xs`. Linux compilers do not accept `-xs={yes|no}`.

3.4.176 **–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 takes effect only when used with optimization level `-xO5` and one of the following `-xarch` values: `sparc`, `sparcvis`, `sparcvis2`, or `sparcvis3` for both `-m32` and `-m64`.

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.

3.4.177 **-xsegment_align=**n

(Oracle Solaris) This option causes the driver to include a special mapfile on the link line. The mapfile aligns the text, data, and bss segments to the value specified by n. When using very large pages, it is important that the heap and stack segments are aligned on an appropriate boundary. If these segments are not aligned, small pages will be used up to the next boundary, which could cause a performance degradation. The mapfile ensures that the segments are aligned on an appropriate boundary.

The n value must be one of the following:

SPARC: The following values are valid: 8K, 64K, 512K, 2M, 4M, 32M, 256M, 1G, and `none`.

x86: The following values are valid: 4K, 8K, 64K, 512K, 2M, 4M, 32M, 256M, 1G, and `none`.

The default for both SPARC and x86 is `none`.

Recommended usage is as follows:

```
SPARC 32-bit compilation: -xsegment_align=64K
SPARC 64-bit compilation: -xsegment_align=4M

x86 32-bit compilation: -xsegment_align=8K
x86 64-bit compilation: -xsegment_align=4M
```

The driver will include the appropriate mapfile. For example, if the user specifies -xsegment_align=4M, the driver adds -M *install-directory*/lib/compilers/mapfiles/map.4M.align to the link line, where *install-directory* is the installation directory. The aforementioned segments will then be aligned on a 4M boundary.

3.4.178 **–xspace**

Do no optimizations that increase the code size.

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

3.4.179 **–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.

The actual expansion of -xtarget values can change from release to release. You can always determine the expansion that the compiler will use with the -dryrun flag:

```
demo% f95 -dryrun -xtarget=ultra4plus
###     command line files and options (expanded):
### -dryrun -xarch=sparcvis
    -xcache=64/32/4/1:2048/64/4/2:32768/64/4/2 -xchip=ultra4plus
```

Note that the –xtarget expansion for a particular named platform might not be the same as –xtarget=native on that same platform.

3.4.179.1   Generic and Native Platforms

native   Optimize performance for the host platform (32–bits).

                        Expands to –m32 –xarch=native –xchip=native –xcache=native

native64   Obsolete. Use –xtarget=native –m64 instead.

generic   Get the best performance for most 32–bit platforms.

                        This is the default and expands to: -m32 -xarch=generic -
                        xchip=generic -xcache=generic

generic64   Obsolete. Use -xtarget=generic -m64 instead.

platform–name   Get the best performance for the specified platform listed below.

3.4.179.2   SPARC Platforms

The following table gives a list of the commonly used system platform names accepted by the
compiler.

TABLE 3-21   Expansions of Commonly Used -xtarget System Platforms

-xtarget= *platform-name*	-xarch	-xchip	-xcache
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10
sparc64viiplus	sparcima	sparc64viiplus	64/64/2:11264/256/11
sparc64x	sparcace	sparc64x	64/128/4/2:24576/128/24/32
sparc64xplus	sparcaceplus	sparc64xplus	64/128/4/2:24576/128/24/32
ultra	sparcvis	ultra	16/32/1:512/64/1
ultra1/140	sparcvis	ultra	16/32/1:512/64/1
ultra1/170	sparcvis	ultra	16/32/1:512/64/1
ultra1/200	sparcvis	ultra	16/32/1:512/64/1
ultra2	sparcvis	ultra2	16/32/1:512/64/1
ultra2/1170	sparcvis	ultra	16/32/1:512/64/1
ultra2/1200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/1300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2/2170	sparcvis	ultra	16/32/1:512/64/1
ultra2/2200	sparcvis	ultra	16/32/1:1024/64/1
ultra2/2300	sparcvis	ultra2	16/32/1:2048/64/1
ultra2e	sparcvis	ultra2e	16/32/1:256/64/4

-xtarget= *platform-name*	-xarch	-xchip	-xcache
ultra2i	sparcvis	ultra2i	16/32/1:512/64/1
ultra3	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4/1:2048/64/4/2:32768/64/4/2
ultraT1	sparc	ultraT1	8/16/4/4:3072/64/12/32
ultraT2	sparcvis2	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
ultraT3	sparcvis3	ultraT3	8/16/4:6144/64/24
T3	sparcvis3	T3	8/16/4:6144/64/24
T4	sparc4	T4	16/32/4:128/32/8:4096/64/16
T5	sparc4	T5	16/32/4/8:128/32/8/8:8192/64/16/128
T7	sparc5	T7	16/32/4/8:256/64/8/16:8192/64/8/32
M5	sparc4	M5	16/32/4/8:128/32/8/8:49152/64/12/48
M6	sparc4	M6	16/32/4/8:128/32/8/8:49152/64/12/96
M7	sparc5	M7	16/32/4/8:256/64/8/16:8192/64/8/32

Note - The following SPARC -xtarget values are obsolete and may be removed in a
future release: ultra, ultra1/140, ultra1/170, ultra1/200, ultra2, ultra2e, ultra2i,
ultra2/1170, ultra2/1200, ultra2/1300, ultra2/2170, ultra2/2200, ultra2/2300, ultra3,
ultra3cu, ultra3i, ultra4, and ultra4plus.

Compiling for a 64-bit Solaris OS on 64–bit enabled platforms is indicated by the -m64 flag. If -
xtarget is specified, -m64 must appear after the -xtarget flag, as in:

```
-xtarget=ultra2   ...   -m64
```

otherwise the default 32–bit memory model will be used.

3.4.179.3 x86 Platforms

The valid -xtarget platform names for x86 systems and their expansions are shown in the
following table.

TABLE 3-22   -xtarget Values on x86 Platforms

-xtarget=	-xarch	-xchip	-xcache
pentium	386	pentium	generic

`-xtarget=`	`-xarch`	`-xchip`	`-xcache`
`pentium_pro`	`pentium_pro`	`pentium_pro`	`generic`
`pentium3`	`sse`	`pentium3`	`16/32/4:256/32/4`
`pentium4`	`sse2`	`pentium4`	`8/64/4:256/128/8`
`opteron`	`sse2a`	`opteron`	`64/64/2:1024/64/16`
`woodcrest`	`ssse3`	`core2`	`32/64/8:4096/64/16`
`barcelona`	`amdsse4a`	`amdfam10`	`64/64/2:512/64/16`
`penryn`	`sse4_1`	`penryn`	`2/64/8:6144/64/24`
`nehalem`	`sse4_2`	`nehalem`	`32/64/8:256/64/8:` `8192/64/16`
`westmere`	`aes`	`westmere`	`32/64/8:256/64/8:30720/64/24`
`sandybridge`	`avx`	`sandybridge`	`32/64/8/2:256/64/8/2:` `20480/64/20/16`
`ivybridge`	`avx_i`	`ivybridge`	`32/64/8/2:256/64/8/2:` `20480/64/20/16`
`haswell`	`avx2`	`haswell`	`32/64/8/2:256/64/8/2:` `20480/64/20/16`

Compiling for 64-bit Solaris OS on 64-bit enabled x86 platform is indicated by the `-m64` flag. For example, compiling with `-xtarget=opteron` is not necessary or sufficient. If `-xtarget` is specified, the `-m64` option must appear after the `-xtarget` flag, as in:

`-xtarget=opteron -m64`

otherwise the compilation will be 32-bit x86.

3.4.180   `-xtemp=`*path*

Equivalent to `-temp`*path*.

3.4.181   `-xthroughput[={yes|no}]`

The `-xthroughput` option tells the compiler that the application will be run in situations where many processes are simultaneously running on the system

If `-xthroughput=yes`, the compiler will favor optimizations that slightly reduce performance for a single process while improving the amount of work achieved by all the processes on the system. As an example, the compiler might choose to be less aggressive in prefetching data. Such a choice would reduce the memory bandwidth consumed by the process, and as such the

process may run slower, but it would also leave more memory bandwidth to be shared among other processes.

The default is -xthroughput=no.

3.4.182 **—xtime**

Synonym for -time.

3.4.183 **—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:16
- 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.

3.4.184 `-xunboundsym={yes|no}`

Specify whether the program contains references to dynamically bound symbols.

`-xunboundsym=yes` means the program contains references dynamically bound symbols.

`-xunboundsym=no` means the program does not contain references to dynamically bound symbols.

The default is `-xunboundsym=no`.

3.4.185 `—xunroll=`*n*

Synonym for `-unroll=`*n*.

3.4.186 `-xvector[=`*a*`]`

Enables automatic generation of calls to the vector library functions or the generation of the SIMD (Single Instruction Multiple Data) instructions on x86 processors that support SIMD. You must use default rounding mode by specifying `-fround=nearest` when you use this option.

The `-xvector` option requires optimization level `-x03` or greater. Compilation will not proceed if the optimization level is unspecified or lower than `-x03`, and a message is issued.

The possible values for *a* are listed in the following table. The `no%` prefix disables the associated suboption.

TABLE 3-23   `-xvector` Suboptions

Value	Meaning
`[no%]lib`	(*Oracle Solaris*) Enables the compiler to transform math library calls within loops into single calls to the equivalent vector math routines when such transformations are possible. This could result in a performance improvement for loops with large loop counts. Use `no%lib` to disable this option.
`[no%]simd`	(*SPARC*) For `-xarch=sparcace` and `-xarch=sparcaceplus`, directs the compiler to use floating point and integral SIMD instructions to improve the performance of certain loops. Contrary to that of the other SPARC platforms, `-xvector=simd` is always effective under `-xarch=sparcace` and `-xarch=sparcaceplus` with the specification of any `-xvector` option, except `-xvector=none` and `-xvector=no%simd`. In addition `-0` greater than 3 is required for `-xvector=simd`, otherwise it is skipped without any warning.

Value	Meaning
	For all other `-xarch` values, directs the compiler to use the Visual Instruction Set [VIS1, VIS2, ViS3, etc.] SIMD instructions to improve the performance of certain loops. Basically with explicit `-xvector=simd` option, the compiler will perform loop transformation enabling the generation of special vectorized SIMD instructions to reduce the number of loop iterations. The `-xvector=simd` option is effective only if `-O` is greater than 3 and `-xarch` is `sparcvis3` and above. Otherwise `-xvector=simd` is skipped without any warning.
`[no%]simd`	(*x86*) Directs the compiler to use the native x86 SSE SIMD instructions to improve performance of certain loops. Streaming extensions are used on x86 by default at optimization level 3 and above where beneficial. Use `no%simd` to disable this option.. The compiler will use SIMD only if streaming extensions exist in the target architecture; that is, if target ISA is at least SSE2. For example, you can specify `-xtarget=woodcrest`, `—xarch=generic64`, `-xarch=sse2`, `-xarch=sse3`, or `-fast` on a modern platform to use it. If the target ISA has no streaming extensions, the suboption will have no effect.
`%none`	Disable this option completely.
`yes`	This option is deprecated; specify `-xvector=lib` instead.
`no`	This option is deprecated; specify `-xvector=%none` instead.

The default is `-xvector=simd` on x86 and `-xvector=%none` on SPARC platforms. If you specify `-xvector` without a suboption, the compiler assumes `-xvector=simd,lib` on x86 Solaris, `-xvector=lib` on SPARC Solaris, and `-xvector=simd` on Linux platforms.

The compiler includes the `libmvec` libraries in the load step.

If you compile and link with separate commands, be sure to use `-xvector` in the linking `CC` command as well.

3.4.187 **—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
```

The options `-ztext` and `-xprofile=collect` should not be used together. While `-ztext` prohibits position-dependent symbol relocations in read-only storage, `-xprofile=collect` generates statically initialized, position-dependent symbol relocations in writable storage.

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.

4

Solaris Studio Fortran Features and Extensions

This chapter describes some of the major language features and extensions in the Oracle Solaris Studio Fortran compiler, `f95`.

4.1 Source Language Features

The `f95` compiler provides the following source language features and extensions to the Fortran standard.

4.1.1 Continuation Line Limits

`f95` allows 999 continuation lines (1 initial and 999 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.

4.1.3 Tab Form

The `f95` fixed-format source text is defined as follows:

- A tab in any of columns 1 through 6 makes the line as a tab form source line.
- A comment indicator or a statement number may precede the tab.
- If a tab is the first nonblank character, then:
    - If the character after the tab is anything other than a nonzero digit, then the text following the tab is an initial line.
    - If there is a nonzero digit after the first tab, the line is a continuation line. The text following the nonzero digit is the next part of the statement.

The `f95` default maximum line length is 72 columns for fixed form and 132 for free form. Use the `-e` compiler option to extend the lines in fixed-format source to 132 columns.

Example: The tab form source on the left is treated as shown on the right.

```
!^IUses of tabs                                  !      Uses of tabs
^ICHARACTER *3 A = 'A'                                  CHARACTER *3 A = 'A'
^IINTEGER B = 2                                         INTEGER B = 2
^IREAL C = 3.0                                          REAL C = 3.0
^IWRITE(*,9) A, B, C                                    WRITE(*,9) A, B, C
9^IFORMAT(1X, A3,                                9      FORMAT(1X, A3,
^I1 I3,                                                 1 I3,
^I2 F9.1 )                                              2 F9.1 )
 ^IEND                                                  END
```

In the example above, "`^I`" stands for the tab character, and the line starting with "1" and "2" are continuation lines. The coding is shown to illustrate various tab situations, and not to advocate any one style.

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.
```

When tab formatting is used with the −`f77` option, there is no 132 character limit to the line length. The line can be much longer.

4.1.4 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

Option	Action
`-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.

4.1.4.1   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.
- In the same program unit, tab form *can* be mixed with free or fixed form.

4.1.4.2   Case

Solaris Studio 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.

4.1.5   Limits and Defaults

- A single Fortran 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 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   Rules Governing Boolean Type

- For masking operations, 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

*dddddd*B, where *d* is any octal digit

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

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

Hexadecimal

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

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

Hollerith

Accepted forms for Hollerith data are:

*n*H…	'…'H	"…"H
*n*L…	'…'L	"…"L
*n*R…	'…'R	"…"R

Above, "…" is a string of characters and *n* is the character count.

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

Examples: Octal and hexadecimal constants.

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

Examples: Octal and hexadecimal in assignment statements.

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

Use of an octal or hexadecimal constant in an arithmetic expression can produce undefined results and do not generate syntax errors.

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, 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

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 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 Data Type	Size	DefaultAlignment	Alignment inCOMMON
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
	4	4	4

Fortran Data Type	Size	DefaultAlignment	Alignment inCOMMON
`REAL*4 X`	8	8	4
`REAL*8 X`	16	8/16	4
`REAL*16 X`			
`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 `-m64`) the default alignment is on 16-byte (rather than 8-byte) boundaries, as indicated by 8/16 in the table. This data type is often referred to as *quad precision*.
- 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.

For example, we can 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.

We can also 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 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 result variable.
- A Cray pointee cannot be a structure name or a structure component.
- The type of a Cray pointee cannot be finalizable.

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 legacy FORTRAN 77, 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 the legacy f77 compiler, a POINTER statement cannot be used as a field declaration.

f95 also allows:

- Either "." or "%"T 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 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*.
- Intrinsic functions do not allow both signed and unsigned integer operands, except for the `MAX` and `MIN` functions, which allow both signed and unsigned integer operands only if there is at least one operand of `REAL` type.
- Unsigned arrays cannot appear as arguments to array intrinsic functions.

4.6 Fortran 200x Features

A number of new features in the Fortran 2003 and Fortran 2008 standards appear in this release of the Oracle Solaris Studio Fortran compiler. For details, refer to the appropriate Fortran standard.

4.6.1 Interoperability with C

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

The Fortran compiler implements the features described in Chapter 15 of the Fortran standard. All intrinsic functions defined in Chapter 15 have been implemented. Fortran also provides facilities for defining derived types and enumerations that correspond to C types, as described in Chapter 4 of the 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
```

The `INTRINSIC` keyword is new in Fortran 2003.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 Fortran 2003 standard.

4.6.3   Command-Line Argument Intrinsics

The Fortran 2003 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 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 2003 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 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

Fortran 2003 extends 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
- polymorphic entities in a `CLASS` type specifier

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 accepts the Fortran 2003 `VALUE` type declaration attribute.

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 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

The Fortran 2003 standard defines a new "stream" I/O scheme. Stream I/O access treats a data file as a continuous sequence of bytes, addressable by a positive integer starting from 1. The data file can be connected for either formatted or unformatted access.

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 IMPORT Statement

The IMPORT statement specified entities in the host scoping unit that are accessible by host association. It is allowed only in an interface body.

4.6.10   Fortran 2003 FLUSH I/O Statement

The f95 compiler accepts the Fortran 2003 FLUSH statement. The FLUSH statement makes data written to an external file available to other processes, or causes data placed in an external file by means other than Fortran to be available to a READ statement.

4.6.11   Fortran 2003 POINTER INTENT Feature

The Fortran compiler now supports the INTENT attribute for POINTER dummy arguments: INTENT(IN), INTENT(OUT), or INTENT(INOUT) may be specified for pointer dummies.

For example,

```
subroutine sub(P)
integer, pointer, intent(in) :: p
...

end
```

The `INTENT` attribute for pointers applies to the pointer and not what it points to, so for `INTENT(IN)` pointers, the following are illegal because they modify the pointer:

```
 p => t
 allocate(p)
 deallocate(p)
```

But the following is legal for `INTENT(IN)` pointers, because it modifies the pointee:

```
 p = 400
```

4.6.12   Fortran 2003 Enhanced Array Constructor

Use of square brackets in place of (`/` and `/`) in array constructors is allowed:

```
X = [ 3.2, 4.01, 6.5 ]
```

The Fortran 2003 standard allows the use of square brackets as array constructors. This can conflict with interval constants. When the square brackets are used without the `-xia` option (or similar options to enable interval arithmetic) they are treated as array constructors. When `-xia` option is used, they are treated as constants. Interval users should continue using (`/` and `/`) array constructors to avoid compilation errors.

The array components within an array constructor can have two forms:

type-spec `::`

or

[*type-spec* `::`] *ac-value-list*

When the optional *type-spec* is present, the types and kinds of the array components do not have to be the same as long as they are type compatible to the *type-spec*.

type-spec can be an intrinsic type or a derived type.

4.6.13   Object-Oriented Fortran Support

Complete support for polymorphism in Fortran 2003 is available.

4.6.14   FINAL Subroutine Support

FINAL subroutines, a Fortran 2003 feature, is supported.

4.6.15   Procedure Pointer Support

Procedure pointers, a Fortran 2003 feature, is supported.

4.6.16   Additional Fortran 2003 and Fortran 2008 Features

Refer to the published Fortran 2003 and Fortran 2008 standards for details.

- 2003 extensions to allocatable arrays — reallocation on assignment, and allocatable scalars.
- 2003 extensions to ALLOCATE/DEALLOCATE statements — ERRMSG and SOURCE.
- 2003 extensions MOVE_ALLOC intrinsic.
- 2003 extensions pointer assignment with remapping.
- 2003 extensions MIN/MAX, MIN/MAXVAL and MIN/MAXLOC with character arguments.
- 2003 intrinsics IS_IOSTAT_END, IS_IOSTAT_EOR, NEW_LINE.
- 2003 intrinsic SELECTED_CHAR_KIND.
- 2003 REAL type for argument COUNT_RATE of intrinsic SYSTEM_CLOCK.
- 2003 new restriction on complex SQRT intrinsic result.
- 2008: Use of null pointers as missing optional arguments.
- 2008 Bit Intrinsics: BGE, BGT, BLE, BLT, DSHIFTL, DSHIFTR, LEADZ, POPCNT, POPPAR, TRAILZ, MASKL, MASKR, SHIFTA, SHIFTL, SHIFTR, MERGE_BITS, IALL, IANY, IPARITY.
- Enhanced structure constructor: using component names to construct a structure constant.
- Enhanced PUBLIC/PRIVATE access control on module derived types and components.
- More Fortran 2008 math intrinsic function support. Most Fortran 2008 math intrinsic functions are now supported except for ERFC_SCALED, NORM2 and some REAL*16 variants on x86 platforms.
- Derived types with no components.
- The KIND argument was added to ICHAR, IACHAR, ACHAR, SHAPE, UBOUND, LBOUND, SIZE, MINLOC, MAXLOC, COUNT, LEN, LEN_TRIM, INDEX, SCAN and VERIFY intrinsics.
- The BACK argument was added to MINLOC and MAXLOC intrinsics.
- New intrinsics FINDLOC and STORAGE_SIZE were added.
- New keywords ERRMSG, SOURCE and MOLD were added to ALLOCATE statement, and ERRMSG was added to DEALLOCATE statement.

- Enumeration with `ENUM`.
- `VOLATILE` keyword.
- `PUBLIC/PRIVATE` accessibility on individual components.
- Public entities of private types.
- Enhanced complex constants.
- Fortran 2003 `ISO_FORTRAN_ENV` module.
- New `KIND=` optional argument to intrinsics.
- Names of length up to 127 characters, except for module names, which are limited to 31 characters.
- `ID=` and `PENDING=` specifier in `INQUIRE` statement.
- `POS=` specifier in data transfer and `INQUIRE` statements.
- `BLANK, DECIMAL, DELIM, PAD, ROUND, SIZE` specifiers.
- `DC, DP, RD, RC, RN, RP, RU, RZ` edit descriptors.
- `INTRINSIC` and `NON_INTRINSIC` keywords in `USE`.
- `IS_IOSTAT_END` and `IS_IOSTAT_EOR` intrinsics.
- Support for deferred-length character declarations. For example, `CHARACTER (LEN=:)`, `POINTER :: STR`.
- Support passing `TARGET` objects to `INTENT(IN)` pointer dummies. This is a Fortran 2008 feature.

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 standard. Some are I/O extensions that appeared in the Fortran 77 compiler, `f77`, and are now part of the Fortran 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.

A complete summary of Fortran directives appears in Appendix C, "Fortran Directives Summary".

4.8.1   Form of Special f95 Directive Lines

f95 recognizes its own special directives in addition to those described in "1.8 Command-Line Help" on page 20. 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 a region of code. Both Sun and Cray style parallelization directives are now deprecated as obsolete. The OpenMP directives and parallelization model are preferred. 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` environment 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:

```
mainmain.oone.modmod_one.otwo.modmod_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 version 7.0 of the Fortran 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 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

Notes: P: The name can be passed as an argument. E: External code for the intrinsic is called at run time.

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

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.

5

FORTRAN 77 Compatibility: Migrating to Solaris Studio Fortran

A separate FORTRAN 77 compiler is no longer provided. The Oracle Solaris Studio Fortran compiler, `f95`, will compile most legacy FORTRAN 77 programs, including programs utilizing non-standard extensions previously compiled by the Sun WorkShop `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 Sun WorkShop `f77` compiler, see the legacy *FORTRAN 77 Language Reference* manual at `http://docs.sun.com/source/806-3594/index.html`.

See Chapter 4, "Solaris Studio Fortran Features and Extensions" for other extensions to the Fortran 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 95/2003. 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 legacy 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$w.d$.Ee. [`-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 "4.4 STRUCTURE and UNION (VAX Fortran)" on page 172)
- 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 compiler 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 occurred 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 accepted by the legacy Sun WorkShop FORTRAN 77 compiler, see the archived *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 the legacy Sun WorkShop FORTRAN 77 compiler, but not in the current `f95` compiler.

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`.

FORTRAN 77 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 Legacy FORTRAN 77-Compiled Routines

- To mix `f77` and `f95` object binaries, link with `f95` compiler and 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 Intrinsics

The Fortran standard supports intrinsic functions that FORTRAN 77 did not have. The full set of Fortran 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 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 "3.4.107 –xalias[=*keywords*]" on page 99. This is discussed with

examples in the chapter on porting "dusty deck" programs in the *Fortran Programming Guide*.

5.5   The **f77** Command

Solaris Studio software no longer includes a separate FORTRAN 77 compiler, f77. Recent releases migrated many FORTRAN 77 features into the Fortran 95 compiler, f95. Much of the functionality of the legacy FORTRAN 77 compiler is now available with the Fortran 95 compiler. Current Solaris Studio compiler releases provide an f77 script, which calls the f95 compiler with an appropriate set of default options. Invoking f77 is equivalent to:

```
f95 -f77 -ftrap=%none
```

If it is necessary to link to library routines that were compiled with a previously released f77 compiler, add -xlang=f77 to the command line. However, if you are compiling and linking in separate steps and explicitly specify -xlang=f77, or -lM77, or -lF77, or -lsunmath, you must link with f95 (or the f77 script) and not with cc or CC. If you are also compiling with the -fast flag, add -ftrap=%none after -fast to retain FORTRAN 77's trapping behavior on arithmetic exceptions since -fast sets the trapping mode to "common".

```
f77 -fast -ftrap=%none
```

Invoking the f77 script issues a message to alert you to the fact that you are using the f95 compiler in -f77 compatibility mode. You can disable this message by adding -errtags=INVOKE to the command line. See "3.4.22 –f77[=*list*]" on page 61 for more information.

A
♦♦♦ APPENDIX A

Runtime Error Messages

This appendix describes the error messages generated by the Fortran 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 f95:

```
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
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

Error	Message
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
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

Error	Message
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
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

Error	Message
1127	unassociated STATUS specifier
1128	unassociated UNFORMATTED specifier
1129	unassociated WRITE specifier
1130	zero length file name
1131	ACCESS specifier is not 'SEQUENTIAL' or 'DIRECT'
1132	ACTION specifier is not 'READ', 'WRITE' or 'READWRITE'
1133	BLANK specifier is not 'ZERO' or 'NULL'
1134	DELIM specifier is not 'APOSTROPHE', 'QUOTE', or 'NONE'
1135	unexpected FORM specifier
1136	PAD specifier is not 'YES' or 'NO'
1137	POSITION specifier is not 'APPEND', 'ASIS', or 'REWIND'
1138	RECL specifier is zero or negative
1139	no record length specified for direct-access file
1140	unexpected STATUS specifier
1141	status is specified and not 'OLD' for connected unit
1142	STATUS specifier is not 'KEEP' or 'DELETE'
1143	status 'KEEP' specified for a scratch file
1144	impossible status value
1145	a file name has been specified for a scratch file
1146	attempting to open a unit that is being read from or written to
1147	attempting to close a unit that is being read from or written to
1148	attempting to open a directory
1149	status is 'OLD' and the file is a dangling symbolic link
1150	status is 'NEW' and the file is a symbolic link
1151	no free scratch file names
1152	specifier ACCESS='STREAM' for default unit
1153	stream-access to default unit
1161	device does not support REWIND
1162	read permission required for BACKSPACE
1163	BACKSPACE on direct-access unit
1164	BACKSPACE on binary unit
1165	end-of-file seen while backspacing
1166	write permission required for ENDFILE
1167	ENDFILE on direct-access unit
1168	stream-access to sequential or direct-access unit
1169	stream-access to unconnected unit

Error	Message
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
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
1217	illegal CORR_ACCTION value
1218	infinite loop occurred due to I/O handler enabled
1220	the number of requested bytes is greater than is supported on the target platform
1221	data in a UNION cannot be read from or written to an incompatible file type
2001	invalid constant, structure, or component name
2002	handle not created

Error	Message
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)
2022	`deadlock in using lock variable` (OpenMP)
2023	`lock not set` (OpenMP)

B

Features Release History

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

B.1   Oracle Solaris Studio 12.4 Fortran Release

The Oracle Solaris Studio Fortran 95 compiler version 8.7 is a component of the Oracle Solaris Studio 12.4 release.

- New `-xarch`, `-xchip`, and `-xtarget` values for Intel Ivy Bridge processor on x86.

- New `-xarch`, `-xchip`, and `-xtarget` values for SPARC T5, M5, M6, and M10+ processors.

- Support for Ivy Bridge assembler instructions.

- Support for Ivy Bridge intrinsic functions, which can be found in *solstudio-install-dir*/`lib/` `compilers/include/cc/immintrin.h`.

- Default value for `-xarch=generic` set to `sse2` for `-m32` on x86.

- Support for `-xlinkopt` on x86. Inter-module, inter-procedural code ordering optimizations for large enterprise applications tuned for modern Intel processors. An up to 5% performance boost over a fully optimized binary can be seen for large applications.

- Enhanced `-xs` option to control the trade-off of executable size versus the need to retain object files in order to debug.

- Support for `-xanalyze` and `-xannotate` on Linux.

- Support for `-fopenmp` as a synonym for `-xopenmp=parallel`.

- Compile time for applications using modules is substantially improved and memory overflows due to module processing are eliminated.

- `#pragma ident` can be used in a source file to identify the source version of the compiled object.

- Support for a deferred type parameter (colon) as the LEN type parameter in a character type used in a declaration. For example:

  ```
  character(LEN=:), pointer :: str
  ```
- Support for procedure pointer.

- Support for the `C_F_PROCPOINTER` function in the Fortran 2003 standard for the `ISO_C_BINDING` module. The `C_FUNLOC` function is extended to allow procedure pointer as argument.

- Support for the Fortran 2003 feature for `ABSTRACT` interface.

- Full support for object-oriented Fortran. Typebound procedures with the `GENERIC`, `DEFERRED`, `NON_OVERRIDABLE`, `PASS`, and `NOPASS` attributes are now allowed.

- Support for the Fortran 2003 feature to enable derived type and generic function to have the same name.

- Support for the Fortran 2008 feature to enable passing `TARGET` objects to `INTENT(IN)` pointer dummies.

- Support for the Fortran 2003 feature to allow finalizing routines in derived types.

- New compiler options:

  - `-fma` enables automatic generation of floating-point fused multiply-add instructions.

  - `-fserialio` specifies that the program does not perform I/O in more than one thread at a time.

  - (x86) `-preserve_argvalues` saves copies of register-based function arguments in the stack.

  - `-xdebuginfo` controls how much debugging and observability information is emitted.

  - `-xglobalize` controls globalization of file static variables but not functions.

  - `-xinline_param` allows for changing the heuristics used by the compiler for deciding when to inline a function call.

  - `-xinline_report` generates a report written to standard output on the inlining of functions by the compiler.

  - `-xipo_build` reduces compile time by avoiding optimizations during the initial pass through the compiler, optimizing only at link time.

  - `-xkeep_unref` keeps definitions of unreferenced functions and variables.

  - `-keepmod` retains a module which is not changed when compiled. The default is `-xkeepmod=yes`, which replaces the old behavior when a new module file is created each time even without any changes from the previous compilation.

  - `-xM` generates makefile dependency automatically. In conjunction with the new `-keepmod=yes` option, it allows the most optimal incremental build on Fortran application using modules.

  - `-xpatchpadding` reserves an area of memory before the start of each function.

  - (Oracle Solaris) `-xsegment_align` causes the driver to include a special mapfile on the link line.

  - `-xthroughput` indicates that the application will be run in situations where many processes are simultaneously running on the system.

  - `-xunboundsym` specifies whether the program contains references to dynamically bound symbols.

- The libraries `libfmaxlai`, `libfmaxvai`, `libfminlai`, `libfminvai`, `libfprodai` and `libfsumai` on SPARC platform have not been used by the Studio Fortran compilers since the Sun Studio 10 release in 2005.

  These libraries will be removed in a future release. At that point, object files and executables produced by Studio compilers earlier than the Sun Studio 10 release can no longer be used and must be recompiled with newer Studio compilers. If you have old object files and executables that require any of these libraries and recompiling is not feasible, you should keep the old compiler installation or copy the specific libraries you need from the old compiler installation into the new compiler installation.

B.2   Oracle Solaris Studio 12.3 Fortran Release

The Oracle Solaris Studio Fortran 95 compiler version 8.6 is a component of the Oracle Solaris Studio 12.3 release.

- The Fortran runtime library now supports sequential-access unformatted records larger than 2GB.

- Support for new SPARC T4 platform: –`xtarget=T4`, –`xchip=T4`, –`xarch=sparc4`

- Support for new x86 Platform Sandy Bridge / AVX: –`xtarget=sandybridge` –`xchip=sandybridge` –`xarch=avx`

- Support for new x86 Platform Westmere / AES: –`xtarget=westmere` –`xchip=westmere` –`xarch=aes`

- New compiler option: –`Xlinker` *arg* passes argument to linker, `ld`(1). Equivalent to –`Wl,`*arg.* ("3.4.104 -Xlinker *arg*" on page 97)

- The OpenMP default number of threads, `OMP_NUM_THREADS` is now 2 (was 1). ("3.4.157 –xopenmp[={parallel|noopt|none}]" on page 138)

- Support for the OpenMP 3.1 shared memory parallelization specifications. ("3.4.157 –xopenmp[={parallel|noopt|none}]" on page 138)

- Use –`library=sunperf` to link to the Sun Performance Library. This obsoletes -`xlic_lib=sunperf`. ("3.4.55 -library=sunperf" on page 77)

- Intrinsic routines `LEADZ`, `POPCNT`, and `POPPAR` previously had the return type as the type of the argument. In this release, to be conforming with the Fortran 2008 standard, the intrinsics return a default integer, regardless of argument type. This introduces a minor incompatibility with previous releases.

- Object Oriented Fortran features related to polymorphism are now supported:

  - Supported OOF features: type extension and polymorphic entities: `CLASS` statement, unlimited polymorphism, `SELECT TYPE` construct, `ABSTRACT` derived type, `EXTENDS_TYPE_OF` and `SAME_TYPE_AS` intrinsics, and sequence type assignments to unlimited pointers.

- Unsupported OOF features: typebound procedures: typebound `PROCEDURE` declaration, `GENERIC`, `DEFERRED`, `NON_OVERRIDABLE`, `PASS`, `NOPASS`.
- Other new F2003/2008 features:
  - Enhanced structure constructor: using component names to construct a structure constant.
  - Enhanced `PUBLIC`/`PRIVATE` access control on module derived types and components.
  - More Fortran 2008 math intrinsic function support. Most Fortran 2008 math intrinsic functions are now supported except for `ERFC_SCALED`, `NORM2` and some `REAL*16` variants on x86 platforms.
  - Derived types with no components.
  - The `KIND` argument was added to `ICHAR`, `IACHAR`, `ACHAR`, `SHAPE`, `UBOUND`, `LBOUND`, `SIZE`, `MINLOC`, `MAXLOC`, `COUNT`, `LEN`, `LEN_TRIM`, `INDEX`, `SCAN` and `VERIFY` intrinsics.
  - The `BACK` argument was added to `MINLOC` and `MAXLOC` intrinsics.
  - New intrinsics `FINDLOC` and `STORAGE_SIZE` were added.
  - New keywords `ERRMSG`, `SOURCE` and `MOLD` were added to `ALLOCATE` statement, and `ERRMSG` was added to `DEALLOCATE` statement.

B.3   Oracle Solaris Studio 12.2 Fortran Release

The Oracle Solaris Studio Fortran 95 compiler version 8.5 is a component of the Oracle Solaris Studio 12.2 release.

- Support for the SPARC VIS3 version of the SPARC-V9 instruction set. Compiling with the `-xarch=sparcvis3` option enables the compiler to use instructions from the SPARC-V9 instruction set, plus the UltraSPARC extensions, including the Visual Instruction Set (VIS) version 1.0, the UltraSPARC-III extensions, including the Visual Instruction Set (VIS) version 2.0 the fused multiply-add instructions, and the Visual Instruction Set (VIS) version 3.0.
- The default for the `-xvector` option has changed on x86-based systems to `-xvector=simd`. Streaming extensions are used on x86-based systems by default where beneficial at optimization level 3 and above. The suboption `no%simd` can be used to disable it. On SPARC-based systems, the default is `-xvector=%none`. See
- Support for the AMD SSE4a instruction set is now available. Compile with the `-xarch=amdsse4a` option.
- The new `-traceback` option enables an executable to print a stack trace if a severe error occurs. This option causes the executable to trap a set of signals and print a stack trace and core dump before exiting. If multiple threads generate a signal, a stack trace will be produced only for the first one. To use traceback, add the `-traceback` option when linking

a program with `f95`, `cc`, or `CC`. For convenience, the option is also accepted at compile-time but is ignored. Using the `-traceback` option with the `-G` option to create a shared library is an error. See "3.4.92 –traceback[={%none|common|*signals_list*}]" on page 92

■ The `-mt` option has been changed to `-mt=yes` or `-mt=no`. The `-mt=yes` option assures that libraries are linked in the appropriate order. See "3.4.60 -mt[={yes|no}]" on page 80

■ The `-xprofile=tcov`option has been enhanced to support an optional profile directory pathname, and possibly the ability to generate `tcov`-compatible feedback data. See "3.4.169 –xprofile=*p*" on page 144

■ The new `-xkeepframe[=[%all,%none]` option prohibits stack related optimizations for the named functions. `%all` prohibits stack related optimizations for all the code. `%none` allows stack related optimizations for all the code. The default is `-xkeepframe=%none`. See "3.4.139 –xkeepframe[=[%all,%none,*name*,no%*name*]]" on page 129

■ Additional F2003 features have been implemented. See "4.6 Fortran 200x Features" on page 174

■ The `IVDEP` directive tells the compiler to ignore some or all loop-carried dependences on array references that it finds in a loop for purposes of optimization. This enables the compiler to perform various loop optimizations that would not be otherwise possible. The `-xivdep` option can be used to disable the `IVDEP` directives, or to determine how the directives should be interpreted. See "2.3.3 IVDEP Directive" on page 36

B.4   Sun Studio 12 Update 1 Fortran Release

■ Object files created by the compilers on the Solaris OS on x86 platforms or the Linux OS are incompatible with previous compiler versions if the application code contains functions with parameters or return values using _m128/_m64 data types. Users with .il inline function files, assembler code, or asm inline statements calling these functions also need to be aware of this incompatibility.

■ New x86 –`xtarget` values `woodcrest, penryn, nehalem`.

■ New SPARC –`xtarget` values `ultraT2plus` and `sparc64vii`.

■ New x86 –`xarch` and –`xchip` values `ssse3, sse4_1, sse4_2, core2, penryn, nehalem, barcelona`.

■ New SPARC –`xarch` and –`xchip` values `sparcima, sparc64vii`, and `ultraT2plus`.

■ The `-xprofile=collect` and `-xprofile=use` options provide improved support for profiling multi-threaded, dynamically linked applications.

■ On Solaris platforms, the `-xpec[=yes|no]` option generates a PEC binary that is recompilable for use with the Automatic Tuning System (ATS).

■ The `-xdepend` option is now implicitly enabled for optimization levels `-xO3` or higher, and is no longer included in the expansion of the `-fast` option.

■ Support for OpenMP 3.0 tasking..

- `-xannotate[=yes|no]` (SPARC platforms only) instructs the compiler to create binaries that can be transformed later by binary modification tools like `binopt`(1).
- Quad precision (REAL*16) is implemented on x86 platforms. REAL*16 is 128-bit IEEE floating point.
- The compiler normally creates temporary files in the `/tmp` directory. You can specify another directory by setting the `TMPDIR` environment variable.
- The behavior of the `cpu_time()` Fortran intrinsic routine is different between Solaris and Linux platforms.
- The Fortran 2003 `IMPORT` statement is implemented.

B.5   Sun Studio 12 Fortran Release

- The Fortran compiler is now available on the following Linux (x86 and x64) distributions: SuSe Linux Enterprise Server 9 with Service Pack 3 (or later), Red Hat Enterprise Linux 4, and other Linux distributions based on the 2.6 kernel (although these are not officially supported).
- Use `-m64` to create 64-bit executables and shared libraries.
- New flags for `-xarch` replace obsolete flags.
- New values for `-xtarget` and `-xchip` provide code generation for the UltraSPARC T2 and SPARC64vi processors.
- New flag `-fma=fused` to enable generation of fused multiply-add instructions on processors that support them.
- New flag `-xhwcprof` enables compiler support for dataspace profiling.
- New flag `-xinstrument` to enable performance analysis by the Thread Analyzer
- `-xregs=frameptr` added to `-fast` on x86.
- Support for interval arithmetic on Solaris x86 platform with the `-xarch=sse2` and `-xia` options.
- Explicit prefetch directives accepted on x86 platforms as well as SPARC platforms. (`-xprefetch=explicit`)
- Default format for debugging information has changed from the "stabs" standard to the "dwarf" standard format. (`-xdebugformat=dwarf`).

B.6   Sun Studio 11 Fortran Release

- **New** `-xmodel` **option:**The new `-xmodel` option lets you specify the kernel, small, or medium memory models on the 64-bit AMD architecture. If the size of your global and static variables exceeds two gigabytes, specify `-xmodel=medium`. Otherwise, use the default `-xmodel=small` setting. See .

- **The** `-xvector` **option extended for x86 SSE2 platforms:**The `-xvector` option enables automatic generation of calls to the vector library functions and/or the generation of the SIMD (Single Instruction Multiple Data) instructions. This option now offers the expanded syntax on x86 SSE2 platforms.

- `STACKSIZE` **environment variable enhanced**:The syntax of the `STACKSIZE` environment variable has been enhanced to include a units keyword.

- `-xpagesize` **options available on x86 platforms:**Options `-xpagesize, -xpagesize_heap,` and `-xpagesize_stack` are now enabled on x86 platforms as well as SPARC. See "3.4.159 –xpagesize=*size*" on page 139.

- **New UltraSPARC T1 and UltraSPARC IV+ targets enabled:**Values for `-xarch, -xchip, -xcache,` and `-xtarget` support new UltraSPARC processors. See "3.4.179 –xtarget=*t*" on page 152.

C

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

C.1   General Fortran Directives

General directives accepted by `f95` are described in .

TABLE C-1   Summary of General Fortran Directives

Format
!$PRAGMA *keyword* (*a* [, *a*] …) [, *keyword* (*a* [, *a*] …)] ,…
!$PRAGMA SUN *keyword* (*a* [, *a*] …) [, *keyword* (*a* [, *a*] …)] ,…
Comment-indicator in column 1 may be c, C, !, or *. (We use ! in these examples. f95 free-format must use !.)

C *Directive*	!$PRAGMA C(*list*) Declares a list of names of external functions as C language routines.
IGNORE_TKR *Directive*	!$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*	!$PRAGMA SUN UNROLL=*n* Advises the compiler that the following loop can be unrolled to a length *n*.
WEAK *Directive*	!$PRAGMA WEAK(*name*[=*name2*]) Declares *name* to be a weak symbol, or an alias for *name2*.
OPT *Directive*	!$PRAGMA SUN OPT=*n* Set optimization level for a subprogram to *n*.

PIPELOOP *Directive*	`!$PRAGMA SUN PIPELOOP[=`*n*`]`
	Assert dependency in loop between iterations *n* apart.
PREFETCH *Directives*	`!$PRAGMA SUN_PREFETCH_READ_ONCE (`*name*`)`
	`!$PRAGMA SUN_PREFETCH_READ_MANY (`*name*`)`
	`!$PRAGMA SUN_PREFETCH_WRITE_ONCE (`*name*`)`
	`!$PRAGMA SUN_PREFETCH_WRITE_MANY (`*name*`)`
	Request compiler generate prefetch instructions for references to name. (Requires `-xprefetch` option, which is enabled by default. Prefetch directives can be disabled by compiling with `–xprefetch=no`. Target architecture must also support prefetch instructions, and the compiler optimization level must be set greater than `–xO2`.)
ASSUME *Directives*	`!$PRAGMA [BEGIN} ASSUME (`*expression* `[,`*probability*`])`
	`!$PRAGMA END ASSUME`
	Make assertions about conditions at certain points in the program that the compiler can assume are true.

C.2 Special Fortran Directives

The following directives are only available with `f95`. See "4.8.2 `FIXED` and `FREE` Directives" on page 183 for details.

TABLE C-2   Special Fortran 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.
IVDEP	`!DIR$ IVDEP`
	Assert that the following `DO`, `FORALL`, or `WHERE` loop is free of loop-carried dependences, and can be optimized. Interpretation determined by the `-xivdep` option. See "2.3.3 `IVDEP` Directive" on page 36

C.3 Fortran OpenMP Directives

The Oracle Solaris Studio Fortran compiler supports the OpenMP 3.1 Fortran API. The `-openmp` compiler flag enables these directives. (See "3.4.157 – xopenmp[={parallel|noopt|none}]" on page 138).

See the *OpenMP API User's Guide* for complete details.

Index

Numbers and Symbols

`!DIR$` in directives, 182
`-xcode`, 110
`-xdebuginfo`, 113
`-xglobalize`, 117
`-xinline_param`, 120
`-xinline_report`, 122
`-xtemp`, 155
`-xvector`, 157
`.mod` file, module file , 184

A

`abrupt_underflow`, 65
address space, 99
aliasing, 99
  `-xalias`, 99
align, 61
  See also data
  `-dalign`, 56
  data in COMMON with `-aligncommon`, 51
alignment of data types, 167
`ALLOCATABLE`
  extensions, 176
analyzer compile option, `xF`, 115
arguments, agreement, `Xlist`, 97
arithmetic *See* floating-point
array bounds checking, 54
`asa`, Fortran print utility, 19
assembly code, 90
ASSUME directive, 34

B

backward compatibility, command-line options, 50

binary I/O, 180
binding, dynamic/shared libraries, 58
Boolean
  constant, alternate forms, 164
  type, constants, 163

C

`C(..)` directive, 30
cache
  padding for, 84
  specify hardware cache, 107
`CALL`
  inlining subprogram calls with `-inline`, 74
case, preserve upper and lower case, 93
cc command-line options
  `-xdebuginfo`, 113
  `-xinline_param`, 120
  `-xinline_report`, 122
`CDIR$` in directives, 182
code size, 152
command-line
  help, 20
command-line options
  `-a` (*obsolete*), 51
  `-aligncommon`, 51
  `-ansi`, 52
  `-arg=local`, 52
  `-autopar`, parallelize automatically, 53
  `-Bdynamic`, 54
  `-Bstatic`, 54
  `-C`, check subscripts, 54
  `-c`, compile only, 55
  `-copyargs`, allow stores to literal arguments, 55
  `-dalign`, 56, 63