FORTRAN 77 4.0 User's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 802-2997-10
Revision A, November 1995

Some of the material in this manual is based on the Bell Laboratories document entitled "A Portable Fortran 77 Compiler," by S. I. Feldman and P. J. Weinberger, dated August 1, 1978. Material on the I/O Library is derived from the paper entitled "Introduction to the f77 I/O Library," by David L. Wasley, University of California, Berkeley, California 94720. Further work was done at Sun Microsystems.

Contents

Figures

Tables

Preface

This preface is organized into the following sections:

Purpose and Audience

This guide shows how to use the SunSoft™ compiler, FORTRAN 77 4.0. It describes the following aspects of this compiler:

- Using the compiler command and options
- Global program consistency checking across routines.
- Using iMPact™ multiprocessor FORTRAN 77 MP
- Debugging FORTRAN 77
- Using IEEE floating point with FORTRAN 77
- Making and using libraries
- Using some utilities and development tools
- Mixing C and FORTRAN 77
- Profiling and tuning FORTRAN 77

This guide is for scientists and engineers with the following background:

- Thorough knowledge and experience with FORTRAN 77 programming
- General knowledge and understanding of some operating system
- Familiarity with the Solaris™ or UNIX® commands `cd`, `pwd`, `ls`, `cat`.

This manual does not teach programming or the FORTRAN 77 language. For details on a language feature or a library routine, see the *FORTRAN 77 4.0 Reference Manual*.

How This Book is Organized

This book is organized as follows.

At the end of the book is an invitation to join the SunPro SIG.

Related Documentation

The following documentation is included with FORTRAN 77:

- Manuals
  - Paper manuals (hard copy)
  - On-line manuals
- On-line `man` pages
- `f77 -help` variations
- On-line `READMEs` directory of information files and feedback form
- SunPro SIG (Sun Programmer Special Interest Group) publications and files

Manuals

On-line Manuals

The *on-line manuals viewing system* displays and searches on-line versions of manuals. It uses dynamically linked cross-references. It is included on the CD-ROM and can be installed to hard disc during installation. Installing and starting it is described in the installation manual.

Related Manuals

The following documents are provided on-line or in hard copy, as indicated.

Title	**Hard Copy**	**On-line**
FORTRAN 77 4.0 User's Guide	X	X
FORTRAN 77 4.0 Reference Manual	X	X
Debugging a Program	X	X
Incremental Link Editor	X	X
Numerical Computation Guide	X	X
What Every Computer Scientist Should Know About Floating-Point Arithmetic		X
Installing SunSoft Developer Products on Solaris	X	X

The following documents are also relevant:

- IEEE and ISO POSIX.1 Standard. See *POSIX Library, page 136*.
- *American National Standard Programming Language FORTRAN*,ANSI X3.9-1978, April 1978, American National Standards Institute, Inc.

`man` *Pages*

A `man` page, short for manual page, is a document about a command, function, subroutine, or collection of such things. It answers the questions “What does it do?” and “How do you use it?”. A `man` page serves two major functions:

- Memory jogger—A `man` page reminds the user of details, such as arguments and syntax. It assumes you knew and forgot, and is not a tutorial.
- Quick reference—A `man` page helps find something fast. It is brief and describes the highlights only. It is a *quick* reference, not a complete reference.

Usage

To display a `man` page on line, use the `man` command.

Example: Display the `f77` `man` page:

```
demo$ man f77
```

Example: Display the `man` page for the `man` command:

```
demo$ man man
```

Example: Display `man` page one-line summaries with key word *xyz*:

```
demo$ man -k xyz
or
demo$ apropos xyz
```

The above commands require the `windex` data base, usually installed by a system administrator; see `-w` for the `catman` (1M) command

Operating System man *Pages and FORTRAN 77* man *Pages*

Some man pages have two versions—one for the operating system and one for FORTRAN 77. The default paths cause man to show the one for FORTRAN 77, but you can direct man to search in the operating system man pages directory first.

Example: One way to display the *operating system* man page for ctime:

```
demo$ man -M /usr/man ctime
```

The man command also uses the MANPATH environment variable, which can determine the set of man pages that are accessed. See man(1).

Related man *Pages*

The following man pages may be of interest to FORTRAN 77 users.

man **Page**	**Contents**
f77(1)	Invoke the FORTRAN 77 compiler
asa(1)	Print files having Fortran carriage-control
dbx(1)	Debug by a command-line-driven debugger
debugger(1)	Debug by a graphical-user-interface debugger
fsplit(1)	Print files having Fortran carriage-control
ieee_flags(3M)	Examine, set, or clear floating-point exception bits
ieee_handler(3M)	Handle exceptions
matherr(3M)	Error handling
ild(1)	Incremental link editor for object files
ld(1)	Link editor for object files
xview(7)	OpenWindows parameters, XView Toolkit programming

`f77 -help` Variations

The following variations are meant to suggest other possibilities.

`f77 -help | more`	The list does *not* scroll off the screen.
`f77 -help | grep "par"`	Show only parallel options.
`f77 -help | grep "lib"`	Show only library options.
`f77 -help | lp`	Print a copy on paper.
`f77 -help >` *MyWay*	Put list onto a file, regroup, reorder, delete, …
`f77 -help | tail`	Show how to send feedback to Sun.
`f77 -xhelp=readme`	Display the on-line `READMEs` file.

READMEs

The `READMEs` directory contains information files that describe the new features, software incompatibilities, software bugs, and information that was discovered after the manuals were printed. The location of this directory depends on the Solaris 1.x/2.x and where your software is installed:

	Standard Installation	**Nonstandard Installation to** /*my*/*dir*/
Solaris 1.x	`/usr/lang/READMEs/`	/*my*/*dir*/`READMEs/`
Solaris 2.x	`/opt/SUNWspro/READMEs/`	/*my*/*dir*/`SUNWspro/READMEs/`

The contents are:

File	**Contents**
`feedback`	email template file for sending feedback comments to Sun
`fortran`	`f77` bugs, new features. behavior changes, documentation errata
`ratfor.ps`	*Ratfor User's Guide*, a PostScript™ file. Print it with `lp` on any PostScript-compatible printer that has Palatino font. View it online with `imagetool`. (*Solaris 1.x*: print with `lpr`; view with `pageview`.)

Sun Programmer Special Interest Group (SIG)

The SIG membership entitles you to other documentation and software. A membership form is included at the very end of this book. See "*Join the SunPro SIG Today*," on page 405.

Conventions in Text

We use the following conventions in this manual to display information.

- We show code listing examples in boxes:

```
    WRITE( *, * ) 'Hello world'
```

- The `plain Courier font` shows prompts, coding, and generally anything that is computer output.
- In dialogs, the **`boldface Courier font`** shows text you type in:

```
demo% echo hello
hello
demo%
```

- *Italics* indicate general arguments or parameters that you replace with the appropriate input. Italics also indicate emphasis.
- The small clear triangle Δ shows a blank space where that is significant:

```
ΔΔ36.001
```

- We generally tag nonstandard features with a small black diamond (♦). A program that uses a nonstandard feature does not conform to the ANSI X3.9-1978 standard, as described in *American National Standard Programming Language* FORTRAN 77, ANSI X3.9-1978, April 1978, American National Standards Institute, Inc., abbreviated as the FORTRAN 77 Standard.
- We usually show FORTRAN 77 examples in tab format, not fixed columns. Also, we use uppercase and lowercase, because any one case is misleading.
- We usually abbreviate FORTRAN 77 as `f77`.

Introduction 1

This chapter is organized into the following sections:

The FORTRAN 77 compiler comes with a programming environment that contains certain operating system calls and support libraries. It integrates with other SunSoft development tools, such as the Debugger, `make`, MakeTool, and SCCS. Some examples assume you have installed the Source Compatibility Package.

The FORTRAN 77 compiler is available in various packages and configurations:

- Alone, or as part of a package, such as the FORTRAN 77 WorkShop™
- With or without the iMPact™ MT/MP multiple processor package

1.1 Operating Environments

Each release of `f77` is available *first* on SPARC systems under the Solaris 2.x operating environment. For information on other current platforms or operating environments, see the `/READMEs/fortran_77` file.

The previous major release was ported to Solaris™ 1.x and to Intel® 80386-compatible computers running Solaris 2.x for x86, and some features remain in this guide identified as being "*Solaris 1.x only*" or "*x86 only*," and sometimes "(*1.x only*)" or "(*x86*)".

Most aspects of FORTRAN 77 under 2.x, 1.x, and x86 are the same, including functionality, behavior, and features.

The iMPact multiprocessor FORTRAN 77 features are available only on SPARC, in Solaris 2.3, and later.

Definitions

The Solaris 2.x operating environment includes, among other components:

- The SunOS™ 5.*x* operating system, which is based on the System V Release 4 (SVR4) UNIX operating system, and the ONC+™ family of published networking protocols and distributed services, including ToolTalk™
- The OpenWindows™ 3.x application development platform

The Solaris 1.x operating environment includes, among other components:

- The SunOS 4.1.x operating system, which is based on the UCB 4.3 BSD operating system
- The OpenWindows 3.x application development platform

Abbreviations

For simplicity:

- Solaris 2.x is an abbreviation for "Solaris 2.3 and later."
- Solaris 1.x is an abbreviation for "Solaris 1.1.3 and later."
- SunOS 5.*x* is an abbreviation for "SunOS 5.3 and later."
- SunOS 4.1.*x* is an abbreviation for "SunOS 4.1.3 and later."

1.2 Standards

This compiler is an enhanced FORTRAN 77 development system which:

- Conforms to the ANSI X3.9-1978 FORTRAN 77 standard and the corresponding International Standards Organization number is ISO 1539-1980. NIST (formerly GSA and NBS) validates it at appropriate intervals.
- Conforms to the standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- Provides an IEEE standard 754-1985 floating-point package.
- Provides support on SPARC® systems for optimization exploiting features of SPARC V8, including the SuperSPARC™ implementation. These features are defined in the *SPARC Architecture Manual: Version 8*.

1.3 Extensions

This FORTRAN 77 compiler provides the following features or extensions:

- Global program checking across routines for consistency of arguments, commons, parameters, etc.
- The iMPact multiprocessor FORTRAN 77 package (*Solaris 2.x, SPARC only*)

  iMPact FORTRAN 77 includes automatic and explicit loop parallelization, is integrated tightly with optimization, and requires a separate license.
- Many VAX® / VMS® FORTRAN 77 5.0 extensions, including:
  - `NAMELIST`
  - `DO WHILE`
  - Structures, records, unions, maps
  - Variable format expressions
- You can write FORTRAN 77 programs with many VMS extensions, such as the following, so that these programs run with the same source code on both SPARC and VAX systems:
  - Recursion
  - Pointers
  - Double-precision complex
  - Quadruple-precision real (*SPARC only*)
  - Quadruple-precision complex (*SPARC only*)

Mixing Languages

On Solaris systems, routines written in C, C++, or Pascal can be combined with FORTRAN 77 programs, since these languages have common calling conventions.

Optimization

`f77` has global, peephole, and potential parallelization optimizations. As a result, you can create FORTRAN 77 applications that execute significantly faster. Benchmarks show that even without parallelization, optimized applications can run significantly faster, with an additional reduction in code size when compared to unoptimized code.

1.4 New Features and Behavior Changes

This section lists the new features and behavior changes.

Features in 4.0 that are New Since 3.0/3.0.1

`f77` 4.0 includes the following features that are new or changed since 3.0/3.0.1:

- The `DOSERIAL` and `DOSERIAL*` parallel directives have been added, and the `DOALL` directive expanded; see “Explicit Parallelization” on page 374.
- A directive for unrolling loops has been added. See “The UNROLL Directive” on page 94.
- The `-I`*dir* option now also affects the `f77 INCLUDE` statement, not only the preprocessor #include directive.
- The Incremental Linker is available. It provides faster linking and speeds up development. See “-xildon” on page 80.
- The `-oldstruct` command-line option has been deleted.
- The following new synonyms have been added: `-xautopar`, `-xdepend`, `-xexplicitpar`, `-xloopinfo`, `-xparallel`, `-xreduction`, and `-xvpara`.
- The `-stackvar` restrictions `EQUIVALANCE`, `NAMELIST`, `STRUCTURE`, and `RECORD` have been removed. See “–stackvar” on page 72.

- New options have been added (and some changed):

Table 1-1 Features in 4.0 that are New since 3.0/3.0.1

`-arg=local`	Pass by value result.
`-copyargs`	Allow assignment to constant arguments.
`-dbl`	Double the default size for integers, reals, and so forth.
`-ext_names=`*e*	Make external names with or without underscores.
`-fns`	Turn on SPARC non-standard floating-point mode *(SPARC, 2.x)*.
`-fround=`*r*	Set the IEEE rounding mode in effect at startup *(SPARC, 2.x)*.
`-fsimple`[=*n*]	Allow levels of simple floating-point model.
`-ftrap=`*t*	Set the IEEE trapping mode in effect at startup *(SPARC, 2.x)*.
`-mp=`*x*	Use either Sun-style or Cray-style MP directives *(SPARC, 2.x)*.
`-O5`	Attempt the highest level of optimization.
`-pad=`*p*	Pad local variables or common blocks
`-vax=`*v*	Specify a choice of VMS features to use.
`-xarch=`*a*	Limit the set of instructions the compiler may use *(SPARC, 2.x)*.
`-xcache=`*c*	Define the cache properties for use by the optimizer *(SPARC, 2.x)*.
`-xchip=`*c*	Specify the target processor for use by the optimizer *(SPARC, 2.x)*.
`-xhelp=`*h*	Show help information for `README` file or for options (flags).
`-xildoff`	Turn off the Incremental Linker *(SPARC, 2.x)*.
`-xildon`	Turn on the Incremental Linker *(SPARC, 2.x)*.
`-xprofile=`*p*	Collect data for a profile or use a profile to optimize *(SPARC, 2.x)*.
`-xregs=`*r*	Specify the usage of registers for the generated code *(SPARC, 2.x)*.
`-xsafe=mem`	Allow compiler to assume no memory-based traps *(SPARC, 2.x)*.
`-xspace`	Do no optimizations that increase the code size *(SPARC, 2.x)*.
`-xtarget=`*t*	Specify target system for instruction set *(SPARC, 2.x)*.
`-ztext`	Do not make the library if relocations remain.

- DO-loop code is now implemented differently to allow better optimization and loop parallelization. Legal DO-loops behave exactly the same as before; however, illegal DO-loops—zero-step, loop variable modified within the loop—may display a different behavior.
- Full 64-bit integers have been added. With `-dbl`, integers not declared with a specified size are turned into full 64-bit integers. See “-dbl” on page 44.
- The following `libV77` library routines: `date`, `mvbits`, `ran`, and `secnds`, are now folded into the `libF77` library. That is, you no longer need to compile with the `-lV77` option to get these routines.
- The `OPEN` statement now contains a new keyword specifier, `ACTION=`*act*, where *act* is `READ`, `WRITE`, or `READWRITE`. See the description in the chapter, “Statements,” in the *FORTRAN 77 4.0 Reference Manual*.

Features in 3.0.1 that are New Since 3.0.

A summary of the new features for 3.0.1 is provided in the following table.

Table 1-2 Features in 3.0.1 that are New since 3.0

New or Changed Feature			User’s Guide	Reference Manual
Ported to Solaris 1.x				
Added global program checking: `-Xlist` (arguments, commons, parameters, …)			*page 83, page 173*	
Improved the `-Xlist` output format			*page 173, …*	
Added the following options:				
	`-nocx`	Smaller executable file—shrink by about 128K bytes (*SPARC only*).	*page 60*	
	`-xlibmopt`	Use a library of selected math routines optimized for performance.	*page 82*	
	`-xnolibmopt`	Reset `-fast` so it does not use the library of selected math routines.	*page 83*	
	`-Zlp`	Prepare code for the loop profiler (*2.x, SPARC only*)	*page 91*	
Improved parallelization— do 25% more loops (private arrays, better fusion, …)			*page 357*	
Documented `TMPDIR` environment variable (for runtime scratch files directory)				*Ch 4,* `OPEN`
Documented the `-unroll=`*n* option to do loop unrolling.			*page 73*	
Libraries: Made extensive bug fixes to the `XView` library bindings			*n/a*	
Added the `Xlib` library bindings to the installation CD			*page 168*	
Added the POSIX library to the installation CD (*2.x only*)			*page 168*	

Features in 3.0 that are New Since 2.0/2.0.1

For 3.0, the major new feature is iMPact multiprocessor FORTRAN 77. A summary of all the new features is provided in the following table.

Table 1-3 Features in 3.0 that are New Since 2.0/2.0.1

New or Changed Feature			User Guide	Reference Manual
Added the optional iMPact multiprocessor FORTRAN 77 package *(2.x, SPARC only)*			*page 357, …*	
Updated `etime`—for multiprocessors: it returns *wall clock* time, and `v(2)` is 0.0 *(Solaris 2.x, SPARC only)*			*n/a*	*Ch. 7, dtime, etime*
Added checking for changing a constant at runtime. Trying to change a constant triggers a segmentation fault (`SIGSEGV`). In previous releases, these codes ran, but some had unpredictable answers and no warning.			*n/a*	*Ch. 4, PARAMETER*
Improved array processing to allow better optimization			*n/a*	
Improved subscript checking of `-C` to check the range on each subscript individually (Previously, it checked the range of the array as a whole.)			*page 189*	
Improved the execution speed for optimized code			*n/a*	
Added the new multi-thread-safe FORTRAN 77 library *(Solaris 2.x, SPARC only)*			*page 168*	
Increased the default limit on number of continuation lines from 19 to 99				*Ch. 1*
Improved compilation speed for:				
	Compiles with no optimization		*n/a*	
	Programs with a large number of symbols		*n/a*	
Eliminated the limit on symbol table size, and changed `-Nn` to do nothing			*page 62*	
Added the following options:				
	`-386`	*Generate code for 80386 (x86 only).*	*page 39*	
	`-486`	*Generate code for 80486 (x86 only).*	*page 39*	
	`-autopar`	*Parallelize automatically (Solaris 2.x, SPARC only).*	*page 40*	
	`-cg92`	*Generate code to run on SPARC V8 architecture (SPARC only).*	*page 43*	
	`-depend`	*Data dependencies, analyze loops (SPARC only).*	*page 45*	
	`-fsimple`	*Simple floating-point model.*	*page 51*	
	`-fstore`	*Force floating-point precision of expressions (x86 only).*	*page 52*	
	`-loopinfo`	*Loop info, show which loops are parallelized (Solaris 2.x, SPARC only).*	*page 58*	
	`-mt`	*Multithread safe libs, use for low level threads (Solaris 2.x, SPARC only).*	*page 59*	
	`-pentium`	*Generate code for pentium (x86 only).*	*page 67*	

Table 1-3 Features in 3.0 that are New Since 2.0/2.0.1 *(Continued)*

New or Changed Feature			**User Guide**	**Reference Manual**
	`-reduction`	*Reduction loops, analyze loops for reduction (Solaris 2.x, SPARC only).*	*page 70*	
	`-stackvar`	*Stack the local variables to allow better optimizing with parallelizing.*	*page 72*	
	`-vpara`	*Verbose parallelization, show warnings (Solaris 2.x, SPARC only).*	*page 74*	
	`-noautopar`, `-nodepend`, `-noexplicitpar`, `-noreduction` (*Solaris 2.x, SPARC only*)		*page 60, …*	
	`-nofstore`	(*x86 only*)	*page 61*	
	`-xa`, `-xcg`*year*, `-xlibmil`, `-xlicinfo`, `-xnolib`, `-xO`[*n*], `-xpg`, `-xsb`, `-xsbfast` `(synonyms for compatibility with C)`		*page 75, …*	
Added parallel directive `C$PAR DOALL`, explicit parallelization (*2.x, SPARC only*)			*page 378*	*Ch. 1, directives*
Deleted the option `-cg87` (*It was present for Solaris 1.x only*).			*page 10*	
Deleted descriptions for the following obsolete options. They do nothing, but they do not break `make` files in this release: `-66`, `-align` *_block_*, `-pipe`, `-r4`, `-w66`			*n/a*	

Other software changes that affect FORTRAN 77 are:

- Using the debugger requires the SC3.0.1 debugger release.
- A new fix-and-continue feature is now in the debugger: to fix a routine, compile only that one routine, then link and run your program.
- In the debugger, watch for any change to the value of a variable.
- Some debugger commands have changed. For a complete list, in dbx, type: `help changes`.
- An optional multiple thread library, `libthread`, is now available from SunSoft.
- For the linker debug aids, see `ld`(1), or try: `-Qoption ld -Dhelp` (Solaris 2.3 only)

Differences for FORTRAN in Solaris 2.x/1.x /x86

Most aspects of FORTRAN under 2.x, 1.x, and x86 are the same, including functionality, behavior, and features. There are some differences, however. The following is a summary of some of those differences:

- Multiprocessor FORTRAN 77 is for Solaris 2.x for SPARC only.
- The POSIX library is for Solaris 2.x only.
- Some options work under Solaris 2.x only:

  `-autopar`, `-dy`, `-dn`, `-explicitpar`, `-G`, `-h`, `-loopinfo`, `-noautopar`, `-nodepend`, `-noexplicitpar`, `-noreduction`, `-reduction`, `-R`, `-vpara`, `-xF`, `-xs`, `-Zlp`, `-Ztha`
- Some options are under Solaris 1.x only:

  `-align, -bsdmalloc`
- Some options are under Solaris x86 only:

  `-386`, `-486`, `-fstore`, `-nofstore`, `-pentium`
- Procedures for building a dynamic shared library differ. See “Dynamic Libraries” on page 158.
- Calls, usage, and return codes of signal handlers differ. See “Exception Handlers and ieee_handler()” on page 226.
- Paths for shared libraries and installation are different. For installation, these paths are:
  - Solaris 2.x: `/opt/SUNWspro/SC4.0`
  - Solaris 1.x: `/usr/lang/SC4.0`

  See also “Search Order for Library Search Paths” on page 151.

Behavior Changes

The behavior of some features has changed.

Sun 4/1xx and Sun 4/2xx Systems

Some older Sun workstations do not work with this compiler.

- **Solaris 1.x**—Applications built with this compiler are incompatible with the Sun 4/1xx and Sun 4/2xx systems under Solaris 1.x.
- **Solaris 2.x**—In principle, applications built with this compiler under Solaris 2.x run on Sun 4/1xx and 4/2xx systems under Solaris 2.x, but *very* slowly.

Upgrading from 3.0

The `-Xlist` option output includes error messages about any inconsistent arguments, commons, parameters, and so forth. Earlier versions of `-Xlist` output did not include these error messages.

The `-Xlist` option output does not include an index. Earlier versions of `-Xlist` output did.

Upgrading from 2.0/2.0.1

If you are upgrading from FORTRAN 77 2.0/2.0.1, the following behavior changes may affect your programs. See also the previous section, “Upgrading from 3.0.”

- Possible slower loading: more global symbols than before

  To provide for the fix-and-continue feature, all local variables are available globally to the debugger in a way that requires that they be loaded at link time. This feature can increase load time.

- Changing a constant

  The 3.0/3.0.1 release improves runtime error checking by preventing the changing of a constant. Trying to change a constant triggers a `SIGSEGV`. In previous releases, such programs did run, but some produced unpredictable answers without warning.

Example: Trying to change a constant:

```
PARAMETER (arg=2.71828)
CALL  sbrtn5 ( arg )
...
END
SUBROUTINE  sbrtn5 ( x )
x = 3.14159
RETURN
END
```

An error message results: `possible attempt to modify constant`.

Workaround:

- General: Do not change a constant. If you must change something, make it a variable, not a constant.
- Specific: In the above example, change the `PARAMETER` statement to a `DATA` statement, that is:

Change:	`PARAMETER (arg=2.71828)`
To:	`DATA arg/2.71828/`

- Number of processors for FORTRAN 77 MP

  The number of processors requested by all programs (users) must not exceed the total number of processors available, otherwise performance could be seriously degraded.

  Example: If there are 4 processors on the system, and if each of three programs requests two processors, performance can be seriously degraded.

- Do not call `alarm()` from an MP program.
- Subscript checking at runtime with `-C`

  The subscript checking has been improved with this release. With `-C`, now each subscript of an array is checked. Before, only the total offset was checked.

Example: A program that ran with no error message, but now displays one:

```
DIMENSION a(10,10)
a(11,1) = 0
END
```

- Debugging FORTRAN programs that use other languages

  If you debug FORTRAN programs that use other languages, you can use the new `dbx language` command.

  Sometimes confusion results about which language `dbx` is debugging. The `language` command can fix both confusions.

  - If `dbx` is confused about the programming language, you can specify the language. Type `language fortran` or `language c`, for example.

    Example: Specifying to `dbx` the programming language:

```
(dbx) language fortran
(dbx)
```

  - If you are confused, ask `dbx` about the language. Type `language`.

    Example: Querying `dbx` which programming language:

```
(dbx) language
fortran
(dbx)
```

- Output from an exception handler is unpredictable

  If you make your own exception handler, avoid doing any FORTRAN 77 output from it. If you must do some, then call abort right after the output. This reduces the risk of a system freeze. FORTRAN I/O from an exception handler amounts to recursive I/O. See the next paragraph.

- Recursive I/O does not work reliably

  If you list a function in an I/O list, and if that function does I/O, then during runtime the execution freezes, or some other unpredictable problem arises. This risk exists independent of parallelization.

Example: Recursive I/O that fails intermittently:

```
PRINT *, x, f(x)
END
FUNCTION f(x)
PRINT *, x
RETURN
END
```

Workaround—Avoid recursive I/O.

- `IOINIT`

  The `IOINIT` routine ignores `CCTL`, `BZRO`, `APND`. There is no workaround.

  The `IOINIT` routine uses a different labeled common, and communicates internal flags to the runtime I/O system. Previous releases put the internal flags into the labeled common:

  `COMMON /IOIFLG/ IEOF, ICTL, IBZR`

  This is not a feature you would use intentionally, but if you had a labeled common named `IOIFLG`, it could result in serious errors.

  The current release uses the labeled common:

  `COMMON /_ _IOIFLG/ IEOF, ICTL, IBZR`

  The two leading underscores take this out of the user name space, so it is safer from accidental errors. Names starting with underscores are reserved for the compiler.

- `dtime` and `etime` in iMPact FORTRAN 77 MP

  `dtime` has always returned the CPU time. In MP, `dtime` returns the sum of all the CPU times, so `dtime` can return an unexpectedly large number. This breaks most megaflops calculations.

  Workaround—Use `etime`, which was changed to return wall clock time in an MP program. Wall clock time does not break most megaflops calculations.

Upgrading from 1.4

If you are upgrading from FORTRAN 77 1.4, the following behavior changes may affect your programs, but see also “Upgrading from 2.0/2.0.1” on page 10” and “Upgrading from 3.0” on page 10.

- Debugging optimized code

  You can now compile with both `-g` and `-O` options. That is, you can now debug with optimized code.
  - If you have make files that rely on `-g` overriding `-O`, then you must revise those make files, because `-g` does not override `-O`.
  - If you have makefiles that check for warning messages, such as: `-g` overrides `-O`, you must revise those makefiles.
  - The combination `-O4 -g` turns off the inlining that you usually get with `-O4`. A warning message is issued.

- Using quadruple precision trigonometric functions

  The precision of PI matches what is used in the expression where PI occurs. It was restricted to 66 bits in the 1.4 release.

- Debugging block data subprograms

  There is a behavior change from FORTRAN 1.4 in debugging block data subprograms.

  Symptom—If you are debugging a main program that uses a block data subprogram, then the debugger cannot find variables that are in the block data subprogram.

  Fix—In the debugger, use the `func` command with the name of the block data subprogram.

  Example: Program with block data:

```
    PROGRAM my_main
    COMMON /stuff/ x, y, z
    PRINT *, x
    END
    BLOCK DATA  init
    COMMON /stuff/ a/1.0/, b/2.0/, c/3.0/
    END
```

To debug the above block data program:

In `dbx`, if you type:
then `dbx` cannot find `a`.

```
(dbx) print a
```

However, if you first type:
followed by:
then `dbx` finds `a`.

```
(dbx) func init
(dbx) print a
```

1.5 Compatibility

The FORTRAN 77 4.0 *source* is compatible with FORTRAN 77 3.0/3.0.1 (or earlier), except for minor changes due to operating system changes and bug fixes.

FORTRAN 77 3.0/3.0.1 to 4.0

Executables (`a.out`), libraries (`.a`), and object files (`.o`) compiled and linked in FORTRAN 77 3.0/3.0.1 under Solaris 2.x are compatible with FORTRAN 77 4.0 under Solaris 2.x.

BCP: Running Applications from Solaris 1.x in 2.x

You must install the Binary Compatibility Package for the executable to run.

Executables compiled and linked in Solaris 1.x do run in Solaris 2.3 and later, but they do not run as fast as when they are compiled and linked under the appropriate Solaris release.

Libraries (`.a`) and object files (`.o`) compiled and linked in FORTRAN 77 2.0.1 under Solaris 1.x are *not* compatible with FORTRAN 77 4.0 under Solaris 2.x.

Application Development in Solaris 2.x for 1.x

Under Solaris 2.x, you can make executables and libraries for Solaris 1.x, but it is not recommended. For the compiler to do this correctly, first install the Binary Compatibility Package. Then, to make it all work, you must:

- Use the Solaris 1.x compiler in BCP mode.
- Use the Solaris 1.x linker (`ld`), with `-qpath` set to the path for the 1.x `ld`.
- Link with the Solaris 1.x libraries. If you receive error messages like: `bad magic number`, check the `-L` options and the `LD_LIBRARY_PATH` environment variable.

See "BCP Mode: How to Make 1.x Applications Under 2.x" on page 105.

1.6 Text Editing

In the Solaris environment, several text editors are available.

`vi` A traditional text editor for source programs is `vi`, the Unix visual display editor. For more information, read the `vi`(1) `man` page.

`textedit` A point-and-click-interface text editor available with OpenWindows.

`xemacs` Xemacs is an Emacs editor that provides interfaces to the selection service and to the ToolTalk™ service.

The SPARCworks package uses these two interfaces to provide simple, yet useful, editor integration with two SPARCworks tools: the SourceBrowser and the Debugger. `xemacs` is available in the SPARCworks package.

1.7 Program Development

The following utilities provide assistance in the development of software programs in FORTRAN 77.

`asa` This utility is a FORTRAN 77 output filter for printing files that have FORTRAN 77 carriage-control characters in column one. The UNIX implementation on this system does not use carriage-control since UNIX systems provide no explicit printer files. Use `asa` when you want to transform files formatted with FORTRAN 77 carriage-control conventions into files formatted according to UNIX line-printer conventions. See `asa`(1).

`fsplit` This utility splits one FORTRAN 77 file of several routines into several files, so that there is one routine per file.

`gprof` This utility profiles by procedure. For Solaris 2.x, when the operating system is installed, `gprof` is included if you do a developer install, rather than an end user install; it is also included if you install the `SUNWbtool` package.

`sbrowser` The SourceBrowser is a source code and call graph browser that finds occurrences of any symbol in all source files, including header files. It is included with `dbx`.

`tcov` This utility profiles by statement.

1.8 Debugging

For debugging, the following utilities are available:

`error` A utility to insert compiler error messages at the offending source file line. For Solaris 2.x, when the operating system is installed, `error` is included if you do a developer install, rather than an end user install; it is also included if you install the `SUNWbtool` package.

`-Xlist` An option to check across routines for consistency of arguments, commons, and so on.

`dbx` An interactive symbolic debugger that understands this FORTRAN 77 compiler.

`debugger` A graphical user interface to the `dbx` debugger.

1.9 Performance Library

The SunSoft Performance Library is a library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.

It is based on the standard libraries `BLAS1`, `BLAS2`, `BLAS3`, `LINPACK`, `LAPACK`, `FFTPACK`, and `VFFTPACK`.

Each subprogram in the SunSoft Performance Library performs the same operation and has the same interface as the standard version, but is generally much faster and sometimes more accurate.

See the `performance_library` information file and the `libsunperf` Reference Manual PostScript files in the `READMEs/` directory.

1.10 Licensing

This compiler uses network licensing, as described in the manual *Installing SunSoft Developer Products (SPARC/Solaris)*.

If you invoke the compiler, and a license is available, the compiler starts. If no license is available, your request for a license is put on a queue, and your compile continues when a license becomes available. A single license can be used for any number of simultaneous compiles by a single user on a single machine.

To run FORTRAN 77 and the various utilities, several licenses may be required, depending on the package you have purchased:

- For FORTRAN 77 4.0, purchase and install a FORTRAN 77 4.0 license.
- For `dbx`, `debugger`, and so forth, purchase and install a SPARCworks (or ProWorks) 4.0 license.
- For the iMPact multiprocessor FORTRAN 77 features, purchase and install a separate iMPact multiprocessor license.

Usually a WorkShop includes all the necessary licenses.

The Compiler 2

This chapter is organized into the following sections:

2.1 Uses of the Compiler

The major use of `f77` is to compile source file(s) to make an executable file.

The generated executable file is an `a.out` file. By default, `f77` automatically invokes a linker.

Other common uses are listed below.

Some other uses of `f77` are:

- Generate an executable for multiple processors, `-autopar`.
- Do *global program checking* across source files and subroutines, `-Xlist`.
- Translate source files to:
  - Relocatable binary (`.o`) files; later they can be linked into an executable (`a.out`) file or static library (`.a`) file
  - A dynamic shared library (`.so`) file, `-G`
- Link `.o` files into an executable load module (`a.out`) file.
- Relink only the changed files, `-xildon`

  The Incremental Link Editor, `ild`, is sometimes used in place of the standard linker, `ld`, for faster development. See “-xildon” on page 80 for more information.
- Prepare for debugging, `-g`.
- Prepare for profiling by statement or procedure, `-pg`.
- Prepare for profiling by parallelized loop, `-Zlp`.
- Show the commands built by the compiler, but do not execute, `-dryrun`.
- Perform a simple check for ANSI standard conformance, `-ansi`.

2.2 A Quick Start

This section provides a quick overview of how to compile and run Fortran programs in a Sun system. It is meant for the experienced user who knows FORTRAN 77 thoroughly (but not necessarily Sun or UNIX versions) and who needs to start writing and running programs immediately.

Using `f77`

Using f77 involves three steps:

1. **Create a FORTRAN 77 source file with a `.f`, `.for`, or `.F` file suffix.**
2. **Compile this source file and link, using the `f77` command.**
3. **Execute the program by typing the name of the executable file.**

Example: This program displays a message on the screen:

```
demo% cat greetings.f
    PROGRAM GREETINGS
    PRINT *, 'Real programmers hack FORTRAN 77!'
    END
demo$
```

Compiling

Example: Compile and link using the `f77` command, as follows:

```
demo% f77 -fast greetings.f
greetings.f:
MAIN greetings:
demo%
```

In the above example, `f77` compiles `greetings.f` and puts the executable code on the `a.out` file.

Running

Example: Run the program by typing `a.out` on the command line:

```
demo% a.out
 Real programmers hack FORTRAN 77!
demo%
```

Renaming the Executables

It is awkward to have the result of every compilation on a file called `a.out`, Moreover, if such a file exists, it is overwritten. For good housekeeping, do one of the following:

- After each compilation, use `mv` to change the name of `a.out`:

```
demo% mv a.out greetings
```

- On the command line, use `-o` to rename the output executable file:

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

The above command places the executable code on the `greetings` file.

Either way, run the program by typing the name of the executable file:

```
demo% greetings
 Real programmers hack FORTRAN 77!
demo%
```

If you are not familiar with the UNIX file system, read Chapter 3, “File System and FORTRAN 77 I/O,” or refer to any introductory UNIX book.

2.3 Compile Command

Before you use any release of f77, it must be installed and licensed. Read the manual, *Installing SunSoft Developer Products (SPARC/Solaris)*.

Command-line Syntax

The syntax of a simple compiler command is as follows:

f77 [*options*] *sfn* …

where *sfn* is a FORTRAN 77 source file name that ends in .f, .F, or .for; *options* is one or more of the compiler options.

Example: A compile command with two files:

```
demo% f77  growth.f  fft.f
```

Example: A compile command, same files, with some options:

```
demo% f77  -g  -u  growth.f  fft.f
```

A *more general* form of the compiler command is:

```
f77  [options]  fn …  [-lx]
```

fn is a file name, not necessarily a name of an f77 source file. See “Command-Line File Names” on page 24.

Compile-Link Sequence

With the above commands, if you successfully compile the files `growth.f` and `fft.f`, the object files, `growth.o` and `fft.o`, are generated, then an executable file is created with the default name `a.out`.

The files, `growth.o` and `fft.o,` are not removed. If there is more than one object file (`.o` file), then the object files are not removed. This protocol results in easier relinking if there is a linking error.

If the compile fails, you receive an error message for each error, and no `a.out` and `.o` files are generated.

The general compiler driver `f77` does the following:

- Calls `f77pass1`, the FORTRAN 77 front end
- Calls the code generator, and optionally the optimizer
- Calls `ld`, the linker, which generates the executable file

`fl.f` → `f77` → `f77pass1` → *optimizer/inliner* → *code generator* → `fl.o` → `ld` → `a.out`

The *optimizer / inliner* is optional.

Command-Line File Names

If a file name in the command line has any of the suffixes: `.f`, `.for`, `.F`, `.r`, `.s`, `.S`, `.il`, or `.o`, then the compiler recognizes it and takes appropriate action. If a file name has some other suffix or no suffix, it is passed to the linker.

Table 2-1 File Name Suffixes FORTRAN 77 Recognizes

Suffix	Language	Action
`.f`	FORTRAN 77	Compile FORTRAN 77 source files, put object files in current directory; default name of object file is that of the source but with `.o` suffix.
`.for`	FORTRAN 77	Same as `.f`.
`.F`	FORTRAN 77	Apply the C preprocessor to the FORTRAN 77 source file before FORTRAN 77 compiles it.
`.r`	Ratfor	Process Ratfor source files before compiling.

Table 2-1 File Name Suffixes FORTRAN 77 Recognizes *(Continued)*

Suffix	Language	Action
`.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 inline expansion code template files. The compiler uses these to expand inline calls to selected routines. Since it's the compiler, not the linker, that does this, be sure to include these `.il` files in the compile command.
`.o`	Object Files	Pass object files through to the linker.

Language Preprocessor

The `cpp` program is the C language preprocessor, which is invoked during the first pass of a FORTRAN 77 compilation if the source file name has the `.F` extension. Its main uses for FORTRAN 77 are for constant definitions and conditional compilation. See `cpp`(1), or *–Dnm*, page 43.

Separate Compiling and Linking

You can compile and link in separate steps, a method you would usually opt for if one of several source files has been changed. This way, you need not recompile all the other source files.

Example: Compile and link in separate steps:

```
demo% f77  -c file1.f  file2.f file3.f    (Make .o files)
demo% f77  file1.o  file2.o file3.o      (Make a.out file)
```

Of course, every file named in the first step (as a `.f` file) must also be named in the second step (as a `.o` file).

Consistent Compiling and Linking

Be consistent with compiling and linking. If you compile and link in separate steps, and you *compile* any subprogram with any of these options, then be sure to *link* with the same options.

–a, –autopar, –cg89, –cg92, –dalign, -dbl, -explicitpar, –f, –fast, -misalign, –p, -parallel, –pg, -r8, -xarch=*a*, -xcache=*c*, -xchip=*c*, xprofile=*p*, -xtarget=*t*, -Zlp, -Ztha

Example: Compile sbr.f with –a and smain.f without it:

```
demo% f77 -c -a sbr.f
demo% f77 -c smain.f
demo% f77 -a sbr.o smain.o    {pass –a to the linker}
```

Unrecognized Arguments

Any arguments f77 does not recognize are taken to be one of the following:

- Linker option arguments
- Names of f77-compatible object programs, maybe from a previous run
- Libraries of f77-compatible routines

The basic distinction is option or non-option:

- Unrecognized *options* (with a -) generate f77 warnings.
- Unrecognized *non-options* (no -) generate no f77 warnings. However, they are passed to the linker and if the linker does not recognize them, they generate linker error messages.

2.4 Option Syntax

Some general guidelines for options are:

- -l*x* is the option to link with library lib*x*.a. It is always safer, but not required, to put -l*x* after the list of file names.
- In general, processing of the compiler options is from left to right, so selective overriding of macros can be done.
  - The above rule does not apply to linker options.
  - The -I, -L, and -R options accumulate, not override

- Square brackets enclose parts of the option that can be omitted. For example, in the `-O[`*n*`]` option, the *n* can be omitted, as in `-O` alone.

Files and results of compilations are linked in the order given to make an executable program, named `a.out` by default, or with a name specified by `-o`.

2.5 Most Useful Options

`f77` has many features (options), but the short list below is a good and adequate start, for the following reasons:

- Most users of `f77` use these options.
- Most `f77` *development* can be (and is) done with only these options.

You may need other options for performance improvement and special problems. The best way to find the option you need is to scan the next section, where all options are grouped by what they do.

Table 2-2 Most Useful Options

Action	**Option**	**Details**
Debug—global program checking across routines for consistency of arguments, commons, …	`-Xlist`	*page 83*
Debug—produce additional symbol table information for the debugger.	`–g`	*page 52*
Performance—invoke the optimizer to produce faster running programs..	`–O[`*n*`]`	*page 63*
Performance—produce reasonably efficient compilation and run times using a selection of options	`–fast`	*page 48*
Bind as dynamic (or static) any library listed later in the command: `-Bdynamic`, `-Bstatic`	`–B`*x*	*page 41*
Library—Allow or disallow dynamic libraries for the entire executable: `-dy`, `-dn` *(Solaris 2.x only)*	`–d`*x*	*page 45*
Compile only—Suppress linking; make a `.o` file for each source file.	`–c`	*page 42*
Output file—Name the final output file *nm* instead of `a.out`.	`–o` *nm*	*page 65*
Profile—Profile by *procedure* for `gprof`.	`–pg`	*page 67*

Check the “*Details*” for risks, caveats, restrictions, interactions, and examples.

2.6 Actions Summary (Actions and Options Sorted by Action)

Check the section, "Options Details (Options and Actions Sorted by Option)," for risks, caveats, restrictions, interactions, and examples.

Debugging Options

For the following debugging options, those that are most useful to the most users are listed first, and then in decreasing order of usefulness.

Table 2-3 Debugging Options

Action	**Option**	**Details**
Compile for use with the debugger.	`–g`	*page 52*
Global program checking (GPC)—arguments, commons, ...	`-Xlist`	*page 83*
Check for subscripts out of range.	`-C`	*page 42*
Undeclared variables—show a warning message.	`-u`	*page 73*
Uppercase identifiers—leave in the original case.	`-U`	*page 73*
Version ID—show ID along with name of each compiler pass.	`-V`	*page 74*
Specify what VMS features to extend.	`-vax=`*v*	*page 74*
Allow debugging by `dbx` without `.o` files *(Solaris 2.x only)*.	`-xs`	*page 86*

Floating-point Options

For the following floating-point options, those with the greatest impact to the most users, and that are easiest to use, are listed first, and then in decreasing order of impact and ease of use.

Table 2-4 Floating-Point Options

Action	**Option**	**Details**
Turn on SPARC nonstandard floating-point (*2.x, SPARC only*).	`-fns`	*page 48*
Set IEEE rounding mode in effect at startup (*2.x, SPARC only*).	`-fround=`*r*	*page 50*
Set IEEE trapping mode in effect at startup (*2.x, SPARC only*).	`-ftrap=`*t*	*page 52*

Library Options

For the following library options, those that are most useful, to the most users, are listed first, and then in decreasing order of usefulness.

Table 2-5 Library Options

Action	**Option**	**Details**
Bind as dynamic or static any library listed later in command.	`–B`*x*	*page 41*
Allow or disallow dynamic libraries for executable (*2.x*).	`–d`*x*	*page 45*
Build a dynamic shared library (*2.x*).	`–G`	*page 52*
Search for libraries in this directory first.	`-L`*dir*	*page 56*
Link with library `lib`*x*.	`-l`*x*	*page 57*
Multithread safe libraries, low level threads (*2.x, SPARC*).	`-mt`	*page 59*
No automatic libraries.	`-nolib`	*page 61*
No inline templates.	`-nolibmil`	*page 62*
No run path in executable (*2.x*).	`-norunpath`	*page 62*
Library—do not make library if relocations remain. (*2.x*)	`-ztext`	*page 92*

Licensing Options

The following options are for licensing.

Table 2-6 Licensing Options

Action	**Option**	**Details**
Do not queue the license request.	`-noqueue`	*page 62*
Show license server user IDs.	`-xlicinfo`	*page 82*

Performance Options

For the following performance options, those with the greatest impact to the most users, and that are easiest to use, are listed first, and then in decreasing order of impact and ease of use.

Table 2-7 Performance Options

Action	**Option**	**Details**
Faster execution—make executable run faster.	`–fast`	*page 48*
Optimize for execution time.	`-O[`*n*`]`	*page 63*
Target—specify target instruction set (*2.x, SPARC*).	`-xtarget=`*t*	*page 87*
Collect or use data for a profile to optimize (*2.x, SPARC*).	`-xprofile=`*p*	*page 84*
Double load—allow `f77` to use double load / store (*SPARC*).	`–dalign`	*page 44*
Arithmetic—use simple arithmetic model.	`-fsimple`	*page 51*
Arithmetic—use SPARC non-standard floating point (*SPARC*).	`-fns`	*page 50*
Inline templates—select best.	`-libmil`	*page 58*
Traps—assume no memory-based traps (*2.x, SPARC only*).	`-xsafe=mem`	*page 86*
Unroll loops—allow optimizer to unroll loops *n* times.	`-unroll=`*n*	*page 73*
Fast math—use special fast math routines (SPARC only).	`-xlibmopt`	*page 82*
Architecture—limit the set of instructions (*2.x, SPARC*).	`-xarch=`*a*	*page 75*
Chip—specify target processor for use by `f77` (*2.x, SPARC*).	`-xchip=`*c*	*page 78*
No fast math—reset `-fast` not to use `-xlibmopt` (SPARC).	`-xnolibmopt`	*page 83*
Data dependencies—analyze loops (*SPARC*).	`-depend`	*page 45*
Inline the specified user routines to optimize for speed.	`-inline=`*rlst*	*page 56*
Do no optimizations that increase code size (*SPARC, 2.x*).	`-xspace`	*page 86*
`malloc`—Use faster `malloc` (*Solaris 1.x*).	`-bsdmalloc`	*page 42*
386—generate code for 80386 (*x86*).	`-386`	*page 39*
486—generate code for 80486 (*x86*).	`-486`	*page 39*
Pentium—generate code for pentium (*x86*).	`-pentium`	*page 67*

Parallelization Options

For the following parallelization options, those with the greatest impact to the most users, and that are easiest to use, are listed first, and then in decreasing order of impact and ease of use.

Table 2-8 Parallelization Options (*SPARC, 2.x*)

Action	**Option**	**Details**
Parallelize loops automatically.	`-autopar`	*page 59*
Parallelize explicitly specified loops.	`-explicitpar`	*page 67*
Parallelize reduction loops.	`-reduction`	*page 70*
Parallelize with `-autopar -explicitpar -depend`.	`-parallel`	*page 67*
Specify the style for MP directives (`cray` or `sun`).	`-mp=`*x*	*page 59*
Prepare loops for profiling parallelization.	`-Zlp`	*page 91*
Show which loops are parallelized, at compile time.	`-loopinfo`	*page 58*
Prepare for thread analyzing by `tha`.	`-Ztha`	*page 93*
Stack local variables to optimize with parallelization.	`-stackvar`	*page 72*
Show warnings about parallelization.	`-vpara`	*page 74*
No automatic parallelization.	`-noautopar`	*page 60*
No `-depend`.	`-nodepend`	*page 61*
No explicit parallelization.	`-noexplicitpar`	*page 61*
No reduction.	`-noreduction`	*page 62*

Profiling Options

For the following profiling options, those that are most useful, to the most users, are listed first, and then in decreasing order of usefulness.

Table 2-9 Profiling Options

Action—Do Profile by:	**Option**	**Details**
Basic block for `tcov`, old style.	`-a`	*page 39*
Procedure for `gprof`.	`-pg`	*page 67*
Procedure for `prof`.	`-p`	*page 65*
Loops for parallelization (*SPARC, 2.x*).	`-loopinfo`	*page 58*
Basic block for `tcov`, new style (*SPARC, 2.x*).	`-xprofile=tcov`	*page 84*

Alignment Options

The following options are for alignment variations.

Table 2-10 Alignment Options

Action	Option	Details
Align on 8-byte boundaries (*SPARC*).	`-f`	*page 47*
Allow for misaligned data (*SPARC*).	`-misalign`	*page 58*
Specify what VMS alignment features to use	`-vax=`*v*	*page 74*
Align a common block on page boundaries. (*1.x, SPARC*).	`–align` *_b_*	*page 40*

Backward Compatibility and Legacy Options

The following options are provided for backward compatibility and certain legacy capabilities.

Table 2-11 Backward Compatibility Options

Action	Option	Details
Allow assignment to constant arguments.	`-copyargs`	*page 43*
External name—make external names without underscores.	`-ext_names=`*e*	*page 47*
Nonstandard arithmetic—allow nonstandard arithmetic.	`–fnonstd`	*page 49*
Host—optimize performance for the host system.	`-native`	*page 60*
Output—use old style list-directed output.	`-oldldo`	*page 65*
`DO` loops—use one trip `DO` loops.	`-onetrip`	*page 65*

Miscellaneous Options

The following miscellaneous options are listed alphabetically by the action they perform, but with *1.x* and *x86* options at the end. The topic of the action is provided in the first word or word phrase.

Table 2-12 Miscellaneous Options

Action	**Option**	**Details**
ANSI conformance—identify many non-ANSI extensions.	`–ansi`	*page 40*
Pass by value result.	`-arg=local`	*page 40*
Compile only, do not make a.out, do not execute.	`-c`	*page 42*
Turn unsized integers into true 64-bit integers.	`-dbl`	*page 44*
Preprocessor—define name for use by preprocessor.	`–D`*name*	*page 43*
Command—show commands built by driver.	`–dryrun`	*page 45*
Line length—extend source length maximum to 132.	`–e`	*page 46*
Preprocessor—use `cpp`.	`–F`	*page 47*
Options—show the list of options. Same as `–help`.	`-help`	*page 49*
Integers, short—make default integer size two bytes.	`-i2`	*page 55*
Integers, standard—make default integer size four bytes.	`-i4`	*page 56*
Inset the `include` path.	`-I`*loc*	*page 54*
Table sizes—reset internal compiler tables.	`-N[cdlnqsx]`*k*	*page 62*
Output—rename file.	`-o` *outfil*	*page 65*
Position-independent code—produce.	`-pic`	*page 68*
Position-independent code—with 32-bit addresses (*SPARC*).	`-PIC`	*page 68*
Pass option list to program.	`-Qoption` *pr op*	*page 68*
`REAL` to `DOUBLE`—interpret `REAL` as `REAL*8`.	`-r8`	*page 70*
Symbol table—strip executable of symbol table.	`-s`	*page 71*
SourceBrowser—compile for the SourceBrowser.	`-sb`	*page 71*
SourceBrowser—compile fast for the SourceBrowser.	`-sbfast`	*page 71*
Quiet compile, prompt only—reduce number of messages.	`-silent`	*page 71*
Assembly source—generate only assembly source code.	`-S`	*page 71*
Reorder functions—enable reordering of functions (*2.x*).	`-xF`	*page 80*
Turns off the incremental linker and forces the use of `ld`.	`-xildoff`	*page 80*
Turns on the incremental linker.	`-xildon`	*page 80*

Table 2-12 Miscellaneous Options *(Continued)*

Action	Option	Details
VMS FORTRAN—include more VMS extensions.	`-xl[d]`	*page 81*
Specify usage of registers for generated code (*SPARC, 2.x*).	`-xregs=`*r*	*page 85*
Warning suppression—do not show warnings.	`-w`	*page 75*
Smaller executable file—shrink by about 128K bytes (*1.x*).	`-nocx`	*page 60*
Force precision of expressions (*x86*).	`-fstore`	*page 52*
No forcing of expression precision (*x86*).	`-nofstore`	*page 61*

2.7 Options Summary (Options and Actions Sorted by Option)

The following table summarizes all options. The option `–help` displays essentially this list, as does the `man f77` command. Check "*Details*" for risks, tradeoffs, side effects, restrictions, interactions, and examples.

Table 2-13 Options Summary

Option	Action	Details
`-386`	*Generate code for 80386 (x86 only).*	*page 39*
`-486`	*Generate code for 80486 (x86 only).*	*page 39*
`-a`	*Profile by basic block for tcov.*	*page 39*
`-align` *_block_*	*Align a common block on a page boundary (Solaris 1.x, SPARC only).*	*page 40*
`-ansi`	*ANSI conformance check—identify many non-ANSI extensions.*	*page 40*
`-autopar`	*iMPact—Parallelize loops automatically (Solaris 2.x, SPARC only).*	*page 40*
`-B`*x*	*Bind as dynamic or static any library listed later in the command.*	*page 41*
`-bsdmalloc`	*Use faster malloc (Solaris 1.x only).*	*page 42*
`-C`	*Subscripts—runtime check for array subscripts out of range.*	*page 42*
`-c`	*Compile only, do not produce executables.*	*page 42*
`-copyargs`	*Allow assignment to constant arguments.*	*page 43*
`-cg89`	*Generate code for generic SPARC architecture (SPARC only).*	*page 42*
`-cg92`	*Generate code for SPARC V8 architecture (SPARC only).*	*page 43*
`-D`*name*	*Preprocessor symbol—define symbol nm for the preprocessor.*	*page 43*
`-dalign`	*Double align—allow f77 to use double-word load/store (SPARC only).*	*page 44*
`-dbl`	*Double the default size for integers, reals, and so forth.*	*page 44*
`-depend`	*Analyze loops for data dependencies (SPARC only).*	*page 45*

Table 2-13 Options Summary *(Continued)*

Option	Action	Details
`-dryrun`	*Show commands built by driver, but do not execute.*	*page 45*
`-d`*x*	*Allow or disallow dynamic libraries for the entire executable (Solaris 2.x only).*	*page 45*
`-e`	*Line length—extend the source line maximum length to 132 columns.*	*page 46*
`-explicitpar`	*iMPact—Parallelize loops explicitly (Solaris 2.x, SPARC only).*	*page 46*
`-ext_names=`*e*	*Make external names with or without underscores.*	*page 47*
`-F`	*Apply the C preprocessor before compiling.*	*page 47*
`-f`	*Align on 8-byte boundaries (SPARC only).*	*page 47*
`-fast`	*Optimize for speed of execution using a selection of options.*	*page 48*
`-flags`	*Synonym for -help.*	*page 49*
`-fnonstd`	*Performance—do nonstandard initialization of floating-point hardware.*	*page 49*
`-fns`	*Use the SPARC nonstandard floating-point mode (SPARC, Solaris 2.x only).*	*page 50*
`-fround=`*r*	*Set the IEEE rounding mode in effect at startup (SPARC, Solaris 2.x only).*	*page 50*
`-fsimple`	*Performance—allow simple floating-point model.*	*page 51*
`-fstore`	*Force precision of floating-point expressions (x86 only).*	*page 52*
`-ftrap=`*t*	*Set floating-point trapping mode in effect at startup (SPARC, Solaris 2.x only)*	*page 52*
`-G`	*Library—build a dynamic shared library (Solaris 2.x only).*	*page 52*
`-g`	*Compile for debugging.*	*page 52*
`-h`*name*	*Library—name the dynamic shared library nm (Solaris 2.x only).*	*page 53*
`-help`	*Options—show a list of option summaries.*	*page 54*
`-I`*loc*	*Add dir to the include file search path.*	*page 54*
`-i2`	*Integers—make the default integer size two bytes.*	*page 55*
`-i4`	*Integers—make the default integer size four bytes.*	*page 56*
`-inline=`*rlst*	*Inline—request inlining of the specified user routines for faster execution.*	*page 56*
`-Kpic`	*Synonym for -pic.*	*page 56*
`-KPIC`	*Synonym for -PIC.*	*page 56*
`-L`*dir*	*Library—search for libraries in the dir directory first.*	*page 56*
`-libmil`	*Inline the selected library routines for optimization.*	*page 58*
`-loopinfo`	*iMPact—Show which loops are parallelized (Solaris 2.x, SPARC only).*	*page 58*
`-l`*x*	*Library—link with library libx.a.*	*page 57*
`-misalign`	*Align—allow for misaligned data (SPARC only).*	*page 58*
`-mp=`*x*	*Specify the style for MP directives (Solaris 2.x, SPARC only).*	*page 59*

Table 2-13 Options Summary *(Continued)*

Option	Action	Details
-mt	*Multithread safe libraries—use for low level threads (Solaris 2.x, SPARC only).*	*page 59*
-native	*Optimize performance for the host system.*	*page 60*
-noautopar	*iMPact—Do not parallelize automatically (Solaris 2.x, SPARC only)..*	*page 60*
-nocx	*Make executable file smaller (SPARC only).*	*page 60*
-nodepend	*Cancel -depend in command line (SPARC only).*	*page 61*
-noexplicitpar	*iMPact—Do not parallelize explicitly (Solaris 2.x, SPARC only).*	*page 61*
-nofstore	*Do not force precision of expression (x86 only).*	*page 61*
-nolib	*Library—Do not link with system libraries.*	*page 61*
-nolibmil	*No inline templates—reset –fast not to include inline templates.*	*page 62*
-noqueue	*License—do not queue a license request.*	*page 62*
-noreduction	*iMPact—do no reduction with parallelization (Solaris 2.x, SPARC only).*	*page 62*
-norunpath	*Library—put no run path in executable (Solaris 2.x only).*	*page 62*
-N[cdlnqsx]*k*	*Table sizes—reset internal compiler tables.*	*page 62*
-O[*n*]	*Performance—optimize for execution time.*	*page 63*
-o *outfil*	*Output file—name the executable file nm instead of a.out.*	*page 65*
-oldldo	*Output—use old list-directed output.*	*page 65*
-onetrip	*DO loops—use one trip DO loops.*	*page 65*
-p	*Profile by procedure for prof.*	*page 65*
-pad[=*p*]	*Insert padding for efficient use of cache.*	*page 65*
-parallel	*iMPact—Parallelize with: -autopar, -explicitpar, -depend (SPARC, 2.x).*	*page 67*
-pentium	*Generate code for Pentium (x86 only).*	*page 67*
-pg	*Profile by procedure for prof.*	*page 67*
-pic	*Library—produce position-independent code for shared library.*	*page 68*
-PIC	*Library—similar to -pic, but with 32-bit addresses (SPARC only).*	*page 68*
-Qoption *pro op*	*Option—pass option list to the program pr.*	*page 68*
-R *list*	*Library—store library paths in executable (Solaris 2.x only).*	*page 69*
-r8	*Set 8 byte default for REAL,INTEGER, and LOGICAL.*	*page 70*
-reduction	*iMPact—do reduction loops (Solaris 2.x, SPARC only).*	*page 70*
-S	*Assembly source—generate and leave only assembly source code.*	*page 71*
-s	*Symbol table—strip the executable file of its symbol table.*	*page 71*
-sb	*SourceBrowser—produce table information for the SourceBrowser.*	*page 71*

Table 2-13 Options Summary *(Continued)*

Option	Action	Details
-sbfast	*Similar to -sb, but faster, and makes no object files.*	*page 71*
-silent	*Show prompt only, reduce number of compiler messages.*	*page 71*
-stackvar	*Allocate local variables on the stack for better optimizing with parallelization.*	*page 72*
-temp=*dir*	*Temporary files—define directory for temporary files.*	*page 73*
-time	*Time for execution—show for each compilation pass.*	*page 73*
-U	*Uppercase identifiers—leave identifiers in the original case.*	*page 73*
-u	*Report undeclared variables.*	*page 73*
-unroll=*n*	*Performance—unroll loops: direct the optimizer on unrolling loops.*	*page 73*
-V	*Version ID—similar to -v, but also show version ID.*	*page 74*
-v	*Show name of each compiler pass.*	*page 74*
-vax=*v*	*Specify some coice of VMS features to use.*	*page 74*
-vpara	*iMPact—show verbose parallelization warnings (Solaris 2.x, SPARC only).*	*page 74*
-w	*Warnings—do not show warnings.*	*page 75*
-xa	*Synonym for -a.*	*page 75*
-xarch=*a*	*Limit the instructions f77 can use (SPARC, Solaris 2.x only).*	*page 75*
-xautopar	*Synonym for -autopar (Solaris 2.x only).*	*page 77*
-xcache=*c*	*Define cache properties for the optimizer (SPARC, Solaris 2.x only).*	*page 77*
-xchip=*c*	*Specify processor for the optimizer (SPARC, Solaris 2.x only).*	*page 78*
-xcg89	*Synonym for -cg89.*	*page 78*
-xcg92	*Synonym for -cg92.*	*page 78*
-xdepend	*Synonym for -depend (Solaris 2.x only).*	*page 79*
-xexplicitpar	*Synonym for -explicitpar (Solaris 2.x only).*	*page 79*
-xF	*Function reorder—allow function-level reordering (Solaris 2.x only).*	*page 80*
-xhelp=*h*	*Show help information for README file or options (flags).*	*page 80*
-xildoff	*Turn off the Incremental Linker. (SPARC, Solaris 2.x only).*	*page 80*
-xildon	*Turn on the Incremental Linker (SPARC, Solaris 2.x only).*	*page 80*
-xinline=*rlst*	*Synonym for -inline=rlst.*	*page 81*
-xl	*Extend the language with more VMS FORTRAN features.*	*page 81*
-xld	*VMS—Debug comments: extended language, VMS, plus debug comments.*	*page 82*
-xlibmil	*Synonym for -libmil.*	*page 82*
-xlibmopt	*Use library of optimized math routines (SPARC only).*	*page 82*

Table 2-13 Options Summary *(Continued)*

Option	**Action**	**Details**
`-xlicinfo`	*License information—show license server user IDs.*	*page 82*
`-Xlist`	*Do global program checking.*	*page 83*
`-xloopinfo`	*Synonym for -loopinfo (Solaris 2.x only).*	*page 83*
`-xnolib`	*Synonym for -nolib.*	*page 83*
`-xnolibmil`	*Synonym for -nolibmil.*	*page 83*
`-xnolibmopt`	*Do not use fast math library (SPARC only).*	*page 83*
`-xO[`*n*`]`	*Synonym for -O[n].*	*page 83*
`-xparallel`	*Synonym for -parallel (Solaris 2.x only).*	*page 83*
`-xpg`	*Synonym for -pg.*	*page 83*
`-xprofile=`*p*	*Collect or use data for profile to optimize (SPARC, Solaris 2.x only).*	*page 84*
`-xreduction`	*Synonym for -reduction (Solaris 2.x only).*	*page 70*
`-xregs=`*r*	*Specify register usage (SPARC, Solaris 2.x only).*	*page 85*
`-xsafe=mem`	*Assume no memory-based traps (SPARC, Solaris 2.x only).*	*page 86*
`-xspace`	*Do not increase code size (SPARC, Solaris 2.x only).*	*page 86*
`-xs`	*Allow debugging by dbx without object (.o) files (Solaris 2.x only).*	*page 86*
`-xsb`	*Synonym for -sb.*	*page 86*
`-xsbfast`	*Synonym for -sbfast.*	*page 86*
`-xtarget=`*t*	*Specify system for optimization (SPARC, Solaris 2.x only).*	*page 87*
`-xtime`	*Synonym for -time.*	*page 91*
`-xunroll=`*n*	*Synonym for -unroll=n.*	*page 91*
`-xvpara`	*Synonym for -vpara (Solaris 2.x only).*	*page 91*
`-Zlp`	*iMPact—prepare for profiling by looptool (Solaris 2.x, SPARC only).*	*page 91*
`-ztext`	*Library—make no library with relocations (Solaris 2.x only).*	*page 92*
`-Ztha`	*iMPact— prepare for Thread Analyzer (Solaris 2.x, SPARC only).*	*page 93*

2.8 Options Details (Options and Actions Sorted by Option)

This section shows all `f77` options, including various risks, restrictions, caveats, interactions, examples, and other details.

-386 Generate code for 80386 (*x86 only*).

Generate code that exploits features available on Intel 80386 compatible processors. The default is `-386`.

-486 Generate code for 80486 (*x86 only*).

Generate code that exploits features available on Intel 80486 compatible processors. The default is `-386`. Code compiled with `-486` does run on 80386 hardware, but it may run slightly slower.

–a Profile by basic block for `tcov`.

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

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

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

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

If you compile and link in separate steps, and you compile with `-a`, then be sure to link with `-a`. You can mix `-a` with `-O`*n*; in some earlier versions `-a` overrode `-O`*n*. For another way, read *Performance Tuning an Application*.

–align _*b*_ Align a common block on a page boundary (*Solaris 1.x, SPARC only*).

Solaris 1.x

For the common block whose FORTRAN name is *b*, align it on a page boundary. Its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. The space is required between `-align` and _*b*_. This is a linker option.

Solaris 2.x

Inert. The `-align` option is for compatibility with older versions, and is an inert option. It is recognized, so it does not break any old `make` files, *but it does not do anything.*

Example: Do page-alignment for the common block named `buffo`:

```
demo% f77 -align _buffo_ growth.f
```

This option applies to *uninitialized* data only. If any variable of the common block is initialized in a `DATA` statement, then the block will not be aligned.

If you do *not* use the `-U` option, then use lowercase for the common block name. If you *do* use `-U`, then use the case of the common block name in your source code.

–ansi ANSI conformance check—identify many non-ANSI extensions.

–arg=local Pass by value result.

When you compile with this option, `f77` uses *copy restore* to retain the association of dummy arguments with the actual arguments between references to functions or subroutines with entry statements.

-autopar iMPact—Parallelize loops automatically (*Solaris 2.x, SPARC only*).

Find and parallelize appropriate loops for multiple processors. Do dependence analysis (analyze loops for inter-iteration data dependence) and do loop restructuring. If optimization is not at `-O3` or higher, raise it to `-O3`.

`-autopar` reduces the utility of debugging (`-g`) in that you cannot print variables with `dbx`, but you can still use the `dbx` `where` command to get a symbolic traceback.

Avoid `-autopar` if you do your own thread management. See note under *–mt*.

The `-autopar` option requires the iMPact FORTRAN 77 multiprocessor enhancement package. To get faster code, this option requires a multiple processor system. On a single-processor system the generated code usually runs slower.

Example: Automatic parallelization (assumes you set number of processors):

```
demo% f77 -autopar any.f
```

Refer to Appendix C, "iMPact: Multiple Processors."

To request a number of processors, at runtime set the `PARALLEL` environment variable. The default is 1. Remember:

- Do not request more processors than are available.
- If `N` is the number of processors on the machine, then for a one-user, multiprocessor system, try `PARALLEL=N-1`.

See Section C.3, "Number of Processors."

If you use `-autopar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe FORTRAN runtime library. If you use `-autopar` and compile and link in *separate* steps, then you must also link with `-autopar`.

–B*x* Bind as dynamic or static any library listed later in the command.

The *x* must be `dynamic` or `static`. No space is allowed between `-B` and `dynamic` or `static`.

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

If you use neither `-Bdynamic` nor `-Bstatic`, the default applies: `dynamic`.

Also note:

- If you specify `static`, but it finds only a dynamic version, then the library is not linked, and you get a warning that the “library was not found.”
- If you specify `dynamic`, but it finds only a static version, then the library is linked, and you get 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.

These are loader and linker options. If you compile and link in separate steps, and you need `-B`*x*, then you need it in the link step.

-bsdmalloc

Use faster `malloc` (*Solaris 1.x only*).

Use the faster `malloc` from the library `libbsdmalloc.a`. This `malloc` is faster but less memory efficient. This option causes the following items to be passed to the linker *(Solaris 1.1.2 and 1.1.3 only)*:

-u _malloc /lib/libbsdmalloc.a

–C

Subscripts—runtime check for array subscripts out of range.

Check for subscripts outside the declared bounds. This helps catch some causes of the dreaded segmentation fault.

If `f77` detects such an out-of-range condition at compile time, it issues an error message and does not make an executable. If `f77` cannot determine such an out-of-range condition until runtime, it inserts range-checking code into the executable. Naturally this option can increase execution time, and the increase may vary from trivial to significant. Some developers debug with `-C`, then recompile without `-C` for the final production executable.

–c

Compile only, do not produce executables.

Suppress linking. Make a `.o` file for each source file.

–cg89

Generate code for generic SPARC architecture (*SPARC only*).

This option is a macro for: `-xarch=v7 -xchip=old -xcache=64/32/1` (*Solaris 2.x only*).

-cg92 Generate code for SPARC V8 architecture (*SPARC* only).

This option is a macro for:
`-xarch=v8 -xchip=super -xcache=16/64/4:1024/64/1` (*Solaris 2.x only*).

-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 for changing a constant.

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

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

–D*nm* Preprocessor symbol—define symbol *nm* for the preprocessor.

`-D`*nm*=*def*

Define *nm* to be *def*

`-D`*nm*

Define *nm* to be `1`

For `.F` files only: Define *nm* to be *def* using the C preprocessor, `cpp`(1), as if by `#define`. If no definition is given, the name is assigned the value 1.

Following are the predefined values:

- The compiler version is predefined (in hex) in `_ _SUNPRO_F77`

  Example: For FORTRAN 77 4.0, `_ _SUNPRO_F77=0x40`
- The following values are predefined on appropriate systems:

  `_ _sparc`, `_ _unix`, `_ _sun`, `_ _i386`, `_ _SVR4`, `_ _SunOS_5_3`

For instance, the value `__i386` is defined on systems compatible with the 80386 (including the 80486), and it is not defined on SPARC systems. You can use these values in such preprocessor conditionals as the following.

```
#ifdef __sparc
```

- From earlier releases, these values (with no underscores) are also predefined, but they may be deleted in a future release:

  `sparc`, `unix`, `sun`, `i386`

–dalign

Double align—allow `f77` to use double-word load / store (*SPARC only*).

Allow `f77` to generate double-word load / store instructions (wherever possible) for faster execution.

Using this option automatically triggers the `–f` option, which causes all double-precision and quadruple-precision data types (both real and complex) to be double aligned.

Using both `-dbl` and `-dalign` also causes 64-bit integer data type to be 8-byte aligned.

With `-dalign`, you may not get ANSI standard FORTRAN 77 alignment. It is a trade-off of portability for speed.

If you compile one subprogram with `-dalign`, compile all subprograms of the program with `–dalign`.

-dbl

Double the default size for integers, reals, and so forth.

With `-dbl`, `f77` sets the default size for `REAL`,`INTEGER`, and `LOGICAL` to 8, and for `COMPLEX` to 16.

For SPARC: `f77` also sets the default size for `DOUBLE PRECISION` to 16, and for `DOUBLE COMPLEX` to 32.

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

`-dbl` allows `INTEGER*8` , but we recommend that you *not* use `INTEGER*8` in your code, since the program will not compile if you omit `-dbl`. Instead, use `INTEGER` (without `*`*n*) and compile with `-dbl`, which automatically converts `INTEGER` to 64-bit integers.

Example: Compile with and without `-dbl`:

```
INTEGER x      {With -dbl, compiles x as 64-bit; without -dbl, compiles x as 32-bit}
INTEGER*8 y    {With -dbl, compiles y as 64-bit; without -dbl, does not compile}
```

Compare `-dbl` with `-r8`:

- For all of the floating point data types, `-dbl` works the same as `-r8`; using both `-r8` and`-dbl` produces the same results as using only `-dbl`.
- For `INTEGER` and `LOGICAL` data types, `-dbl` is different from `-r8`:
  - With `-dbl`, `f77` allocates 8 bytes, and does 8-byte arithmetic
  - With `-r8`, `f77` allocates 8 bytes, and does only 4-byte arithmetic

-depend Analyze loops for data dependencies (*SPARC only*).

Analyze loops for inter-iteration data dependencies and do loop restructuring. The `-depend` option is ignored unless you also use `-O3` or `-O4`. Dependence analysis is also included with `-autopar` or `-parallel`. The dependence analysis is done at compile time.

The iMPact FORTRAN 77 multiprocessor package is *not* required for `-depend`.

–dryrun Show commands built by driver, but do not execute.

–d*x* Allow or disallow *dynamic* libraries for the entire executable (*Solaris 2.x only*).

The *x* must be `y` or `n`. No space is allowed between `-d` and `y` or `n`.

- `-dy`: Yes—allow *dynamically* bound libraries (*allow* shared libraries).
- `-dn`: No—do *not* allow dynamically bound libraries (*no* shared libraries).

If you have neither `-dy` nor `-dn`, you get the default: `-dy`.

These apply to the *whole* executable. Use only *once* on the command line.

If `a.out` uses *only static* libraries, then `-dy` causes a few seconds delay at runtime it makes the *dynamic* linker be invoked when `a.out` is run. This takes a few seconds to invoke and find that no dynamic libraries are needed.

`-d`*binding* is a loader and linker option. If you compile and link in separate steps, and you need `-d`*binding*, then you need it in the link step.

–e Line length—extend the source line maximum length to 132 columns.

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

-explicitpar iMPact—Parallelize loops explicitly (*Solaris 2.x*, *SPARC only*).

You do the dependency analysis: analyze and specify loops for inter-iteration data dependencies. The software parallelizes the specified loops. If optimization is not at `-O3` or higher, then it is raised to `-O3`. Avoid `-explicitpar` if you do your own thread management. See *–mt*.

`-explicitpar` reduces the utility of debugging (`-g`) in that you cannot print variables, but you can use the `dbx where` command to get a symbolic traceback.

The `-explicitpar` option requires the iMPact FORTRAN 77 multiprocessor enhancement package. To get faster code, this option requires a multiprocessor system. On a single-processor system the generated code usually runs slower.

Refer to Appendix C, "iMPact: Multiple Processors."

Summary: To parallelize explicitly, do the following:

1. **Analyze the loops to find those that are safe to parallelize.**
2. **Insert** `C$PAR DOALL` **immediately before the safe loops.**
3. **Compile with the** `-explicitpar` **option.**

Example: Insert a parallel directive immediately before the loop:

```
       ...
C$PAR DOALL
    do i = 1, n
   a(i) = b(i) * c(i)
    end do
       ...
```

Example: Compile to explicitly parallelize:

```
demo% f77 -explicitpar any.f
```

Do *not* apply an explicit parallel directive to a reduction loop. The explicit parallelization is done, but the reduction aspect of the loop is not done, and the results can be incorrect. The results of the calculations can even be *indeterminate*: you can get incorrect results, possibly different ones with each run, and with no warnings.

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

-ext_names=*e* Make external names with or without underscores.

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

`plain`: Do not use trailing underscores.

`underscores`: Use trailing underscores.

An external name is a name of a subroutine, function, block data subprogram, or labeled common. This option affects both the name in the routine itself and, of course, the name used in the calling statement (both symdefs and symrefs).

–F Apply the C preprocessor before compiling.

Apply the cpp preprocessor to `.F` files and put the result in the file with the suffix changed to `.f`, but do not compile.

–f Align on 8-byte boundaries (*SPARC only*).

Align all `COMMON` blocks and all double-precision and quadruple-precision local data on 8-byte boundaries. This applies to both real and complex data.

Using both `-dbl` and `-f` also causes 64-bit integer data type to be 8-byte aligned.

Resulting code may not be standard and may not be portable.

If you compile with `-f` for *any* subprogram of a program, then compile *all* subprograms of that program with `-f`.

–fast Optimize for speed of execution using a selection of options.

Select options that optimize for speed of execution without excessive compilation time. This option provides close-to-the-maximum performance for many applications.

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

Note – Details of what `-fast` provides vary with the compiler. See the documentation about C, C++, FORTRAN 77, Fortran 90, or Pascal for specifics.

`-fast` selects the following options:

- The `-native` best floating-point option

  If the program is intended to run on a different target than the compilation machine, follow the `-fast` with a code-generator option. For *SPARC*, an example is: `-fast -cg89`

- The `–O3` optimization level option

  For subprograms that benefit from more optimization, follow `–fast` with `-O4` or `-O5`: `-fast -O4`. Using `-fast -O4` pair is not the same as using the `-O4 -fast` pair. The last specification of each pair takes precedence.

  Example: Overriding part of `-fast` (note warning message):

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

- The `-libmil` option for system-supplied inline expansion templates

  For C functions that depend on exception handling specified by SVID (as do some `libm` programs), follow `-fast` by `-nolibmil`: `-fast –nolibmil`. With `-libmil`, exceptions cannot be detected with `errno` or `matherr`(3m).

- The `-fsimple` option for a simple floating-point model

  `–fsimple` is unsuitable if strict IEEE 754 standards compliance is required.
- The `–dalign` option to generate double loads and stores (*SPARC only*)

  Using this option may not generate the ANSI standard FORTRAN 77 alignment.
- The `-xlibmopt` option (*SPARC only*)
- For x86 only: The `-nofstore` option, so it does not force floating-point expressions to the precision of the destination variable.
- For Solaris 2.x only: `-fns -ftrap=%none`; that is, turn off all trapping. In previous releases, the `-fast` macro option included `-fnonstd;` now it does not.

`-flags`

Synonym for `-help`.

`–fnonstd`

Performance—do nonstandard initialization of floating-point hardware.

Do nonstandard initialization of floating-point arithmetic hardware:

- Abort on exceptions
- Flush denormalized numbers to zero if it will improve speed

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

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 denormalized numbers.

The standard initialization of floating-point is the default:

- IEEE 754 floating-point arithmetic is nonstop.
- Underflows are gradual.

Specifying –fnonstd during the link step is approximately equivalent to the following two calls at the beginning of a FORTRAN 77 main program.

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

The nonstandard_arithmetic() routine is equivalent to the obsolete abrupt_underflow() routine.

On some floating-point hardware, the nonstandard_arithmetic() call causes all underflows to produce zero rather than a possibly subnormal number, as the IEEE standard requires. This may be faster. See ieee_functions(3m).

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 to a core file. See ieee_handler(3m).

This option is a synonym for -fns -ftrap=common (*Solaris 2.x only*).

-fns Use the SPARC nonstandard floating-point mode (*SPARC, Solaris 2.x only*).

The default is the SPARC standard floating-point mode.

If you compile one routine with -fns, then compile all the program routines with the –fns option; otherwise, you can get unexpected results.

-fround=*r* Set the IEEE rounding mode in effect at startup (*SPARC, Solaris 2.x only*).

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

The default is -fround=nearest.

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.

The meanings are the same as those for the ieee_flags function.

If you compile one routine with -fround=*r*, compile all the program routines with the same –fround=*r* option; otherwise, you can get unexpected results.

-fsimple[=*n*] Performance—allow simple floating-point model.

Allow the optimizer to make simplifying assumptions concerning floating-point arithmetic.

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

- If there is no `-fsimple[=n]` then the compiler uses `-fsimple=0`
- If there is only `-fsimple` then the compiler uses `-fsimple=1`

`-fsimple=0`

Permit no simplifying assumptions. Preserve strict IEEE 754 conformance.

`-fsimple=1`

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

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

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

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

`-fsimple=2`

Permit aggressive floating point optimizations that may cause many programs to produce different numeric results due to changes in rounding.

For example, in a given loop, permit the optimizer to replace all computations of `x/y` with `x*z`, where `z=1/y`, `x/y` is guaranteed to be evaluated at least once in the loop, and the values of `y` and `z` are known to have constant values during execution of the loop.

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

-fstore

Force precision of floating-point expressions (*x86 only*).

Use the precision of destination variable. This option applies to assignment statements only. Unless `-fast` is on, the default is `-fstore`.

-ftrap=*t*

Set floating-point trapping mode in effect at startup (*SPARC, Solaris 2.x only*)

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`.

The default is `-ftrap=%none`. Where the `%` is shown, it is a required character.

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

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

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

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

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you can get unexpected results.

–G

Library—build a dynamic shared library (*Solaris 2.x only*).

Direct the linker to build a *shared dynamic* library. Without `-G`, the linker builds an executable file. With `-G`, it builds a dynamic library. This option does not automatically turn on `-ztext` as it did in the previous release.

–g

Compile for debugging.

Produce additional symbol table information for the debuggers, `dbx`(1) and `debugger`(1).

If you plan to debug, you get more debugging power if you compile with `-g` before using the debuggers. The `–g` option suppresses the automatic inlining you usually get with `–O4`, but does not suppress `–O`*n*.

For *SPARC, Solaris 2.x*: The `-g` option makes `-xildon` the default incremental linker option (see “-xildon” on page 80). That is, with `-g`, the compiler default behavior is to automatically invoke `ild` in place of `ld`, unless the `-G` option is present, or any source file is named on the command line.

The utility of debugging is reduced when options `-autopar`, `-explicitpar`, or `-parallel` are used with `-g` in that you cannot print variables with `dbx`, but you can still use the `dbx where` command to get a symbolic traceback.

`–O`*n* (and parallelization) limits `–g` in the following ways:

- Local variables cannot be printed (optimizer can put them on the stack)
- Cannot step through a routine line by line (optimizer can change the order)

You can get around some `-O`*n* limits on `-g` in either of two ways:

- Recompile all routines with `-O1` or no `-O`*n* at all
- Recompile only the routine you need to debug using *fix and continue*

If you are upgrading from prior to 2.0, note the following side effects of `-g` not suppressing `–O`*n*:

- Old makefiles that rely on `-g` overriding `-O` must be changed.
- Old makefiles that check for the warning: `-g overrides -O`, must be changed.

–h*nm* Library—name the dynamic shared library *nm* (*Solaris 2.x only*).

The `-h`*nm* option assigns a name to a shared dynamic library, and allows versions of a shared dynamic library. A space between `-h` and *nm* is optional. In general, *nm* must be the same as what follows the `-o`.

This is a loader option. The compile-time loader assigns the specified name to the shared dynamic library being created, and it records the name in the library file as the *internal* name of the library.

If there is no `-h`*nm* option, then no internal name is recorded in the library file. Every executable file has a list of needed shared library files. When the runtime linker links the library into an executable file, the linker copies the internal name from the library into that list of needed shared library files.

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

Example: One way to use the `-h` option:

1. Make and use one version of a shared library.

```
demo% ld -G -o libxyz.1 -h libxyz.1 ...  Create shared library
demo% ln libxyz.1 libxyz.so              Make link libxyz.so to libxyz.1
demo% f77 ...-o verA -lzyz  ...          Executable verA needs libxyz.1
```

2. Make and use a second version of the library.

```
demo% ld -G -o libxyz.2 -h libxyz.2 ...  Create shared library
demo% rm libxyz.so                       Remove old link
demo% ln libxyz.2 libxyz.so              Make link libxyz.so to libxyz.2
demo% f77 ...-o verB -lxyz  ...          Executable verB needs libxyz.2
```

–help Options—show a list of option summaries.

Show a of this list of option summaries and show how to send feedback comments to Sun. See also *-xhelp=h*.

–I*dir* Add *dir* to the include file search path.

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

The *include file search path* is the list of directories searched for include files. This search path is used by:

- The preprocessor directive `#include`
- The `f77` statement `INCLUDE`

Example: Search for include files in /usr/applib:

```
demo% f77 -I/usr/applib growth.F
```

Remarks

- For preprocessor `#include`, use `.F`
- For `f77` language `INCLUDE`, use `.f` or `.F`
- Do not use an `INCLUDE` statement to include a `#include` file.
- Use `-I`*dir* again for more paths. Example: `f77 -Ipath1 -Ipath2 any.F`.

Order

The search order for relative path names is:

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

The default list for `–I`*dir* depends on Solaris 1.x/2.x and the directory for `f77` installation. This list is usually set to the following list of paths:

Table 2-14 Default Search Paths for Include Files

	Standard Install	**Nonstandard Install to** /*my*/*dir*/
Solaris 1.x	/usr/lang/SC4.0/include/f77 /usr/include	/*my*/*dir*/SC4.0/include/f77 /usr/include
Solaris 2.x	/opt/SUNWspro/SC3.0.1/include/f77 /usr/include	/*my*/*dir*/SUNWspro/SC3.0.1/include/f77 /usr/include

–i2 Integers—make the default integer size two bytes.

Make two the default size in bytes for integer and logical constants and variables. But for `INTEGER*`*n* `Y`, `Y` uses *n* bytes, regardless of the `–i2`. This option may increase runtime. If you need short integers, it is generally better to use `INTEGER*2` for specific (large) arrays.

-i4 Integers—make the default integer size four bytes.

Make four the default size in bytes for integer and logical constants and variables. In `INTEGER Y`, `Y` uses four bytes, but in `INTEGER*`*n* `Y`, `Y` uses *n* bytes, regardless of `-i4`.

-inline=*rlst* Inline—request inlining of the specified user routines for faster execution.

Request that the optimizer inline the user-written routines named in *rlst*. The list is a comma-separated list of functions and subroutines.

Example: Inline the routines *sub1*, *sub6*, *sub9*:

```
demo% f77 -O3 -inline=sub1,sub6,sub9 *.f
```

Following are the restrictions; no warnings are issued:

- Optimization must be `-O3` or greater *(SPARC, Solaris 2.x)*.
- The source for the routine must be in the file being compiled.
- `f77` decides which ones to inline (inlining must look profitable and safe).

Note the interactions:

- If you compile with `-O3`, the `-inline` option can increase speed by inlining some routines. The `-O3` option inlines none by itself.
- If you compile with `-O4`, the `-inline` can decrease speed by restricting inlining to only those routines in the list. With `-O4`, `f77` normally tries to inline all appropriate user-written subroutines and functions.

-Kpic Synonym for `-pic`.

-KPIC Synonym for `-PIC`.

-L*dir* Library—search for libraries in the *dir* directory first.

Add *dir* at the *start* of the list of object-library search directories. While building the executable file, `ld`(1) searches *dir* for archive libraries (`.a` files) and shared libraries (`.so` files). A space between `-L` and *dir* is optional. The directory *dir* is not built in to the `a.out` file. See also *–lx*. `ld` searches *dir* before the default directories. See “Search Order for Library Search Paths” on page 151. For the relative order between `LD_LIBRARY_PATH` and `-L`*dir*, see `ld`(1).

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

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

Example: Use `-L`*dir* again to add more directories:

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

Here are the restrictions:

Solaris 1.x and 2.x

- No `-L/usr/lib`: In Solaris 1.x and 2.x, do *not* use `–L`*dir* to specify `/usr/lib`. It is searched by default. Including it here may prevent using the unbundled `libm`.

Solaris 2.x

- No `-L/usr/ccs/lib`: In Solaris 2.x, do not use `–L`*dir* to specify `/usr/ccs/lib`. It is searched by default. Including it here may prevent using the unbundled `libm`.

`–l`*x*

Library—link with library `lib`*x*`.a`.

Pass “`-l`*x*” to the linker. `ld` links with object library `lib`*x*. If shared library `lib`*x*`.so` is available, `ld` uses it, otherwise, `ld` uses archive library `lib`*x*`.a`. If it uses a shared library, the name is built in to `a.out`. No space is allowed between `-l` and *x* character strings.

Example: Link with the library `libV77`:

```
demo% f77 any.f –lV77
```

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

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

```
demo% f77 any.f –ly -lz
```

See also "Library Search Paths and Order" on page 149.

-libmil

Inline the selected library routines for optimization.

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

-loopinfo

iMPact—Show which loops are parallelized (Solaris *2.x, SPARC only*).

Show which loops are parallelized and which are not. Use with the `-parallel`, `-autopar`, or `-explicitpar` options.

This option requires the iMPact FORTRAN 77 multiprocessor enhancement package.

Pass `f77` standard error into the `error` utility to get an annotated source listing (each loop tagged as parallelized or not); otherwise, loops are identified only by line number.

Example: `-loopinfo`, in `sh`, pass `f77` standard error to the `error` utility:

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

For details on `error`, see Section 7.5, "Compiler Messages in Listing (error)."

-misalign

Align—allow for misaligned data (*SPARC only*).

The `-misalign` option allows for misaligned data in memory. Use this option *only* if you get a warning that `COMMON` or `EQUIVALENCE` statements cause data to be misaligned. This option generates much slower code for references to dummy arguments. If you can, recode the indicated section instead of recompiling with this option.

Example: The following program has misaligned variables.

```
      INTEGER*2   I(4)
      REAL        R1, R2
      EQUIVALENCE (R1, I(1)), (R2, I(2))
      END
```

The following error message is issued:

```
"misalign.f", line 4: Error: bad alignment for "r2" forced by
equivalence
```

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

–mp=*x*

Specify the style for MP directives (*Solaris 2.x, SPARC only*).

x is either `sun` or `cray`. The default is `sun`. Use only one in any given run.

`-mp=sun`: Accept only the Sun-style MP directives.

These directives start with the `C$PAR` or `!$PAR` prefix.

`-mp=cray`: Accept only the Cray-style MP directives.

These directives start with the `CMIC$` or `!MIC$` prefix.

–mt

Multithread safe libraries—use for low level threads (*Solaris 2.x, SPARC only*).

Use multithread-safe libraries. If you do your own low-level thread management, this option helps prevent conflicts between threads. For FORTRAN 77, the multithread-safe library is `libF77_mt`.

Use `-mt` if you mix C and FORTRAN 77, and you manage multithread C coding using the `libthread` primitives. Before you use your own multi-threaded coding, read the Solaris manual, *Multithreaded Programming Guide*.

The `–mt` option does not require the iMPact FORTRAN 77 multiprocessor enhancement package, but to compile and run it does require Solaris 2.2 or later. The equivalent of `-mt` is included automatically with `-autopar`, `-explicitpar`, or `-parallel`.

On a single-processor system, the generated code can run more slowly with the `–mt` option, but usually not by a significant amount.

The restrictions are:

- With `-mt`, if a function does I/O, do not name that function in an I/O list. Such I/O is called *recursive* I/O, and it causes the program to hang (deadlock). Recursive I/O is unreliable anyway, but is more apt to hang with `-mt`.
- In general, do *not* combine your own multi-threaded coding with `-autopar`, `-explicitpar`, or `-parallel`. Either do it all yourself or let the compiler do it. You may get conflicts and unexpected results if you and the compiler are both trying to manage threads with the same primitives.

–native

Optimize performance for the host system.

The `-fast` option includes `-native` in its expansion.

For Solaris 1.x

Direct the compiler to decide which floating-point options are available on the machine the compiler is running on, and generate code for the best one. If you compile and link in separate steps, and you compile with the `-native` option, then be sure to link with `-native`.

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

For Solaris 2.x

This option is a synonym for `-xtarget=native`.

-noautopar

iMPact—Do not parallelize automatically (*Solaris 2.x, SPARC only*).

Do not parallelize loops automatically. This option requires the FORTRAN 77 multiprocessor enhancement package.

-nocx

Make executable file smaller (*SPARC only*).

This makes it smaller by about 112K bytes. The smaller files are from not linking with `-lcx`. The runtime performance and accuracy of binary-decimal base-conversion will be somewhat compromised.

-nodepend Cancel `-depend` in command line (*SPARC only)*.

Cancel any `-depend` from earlier on the command line. This option does not require the iMPact FORTRAN 77 multiprocessor package.

-noexplicitpar iMPact—Do not parallelize explicitly (*Solaris 2.x*, *SPARC* only).

This option requires the iMPact FORTRAN 77 multiprocessor package.

-nofstore Do not force precision of expression *(x86 only*).

Do not force expression precision to precision of destination variable (*x86 only*). This option is for assignment statements only, and is the default if `-fast` is on.

-nolib Library—Do not link with system libraries.

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

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

For example, a program linked dynamically with `libF77` fails on a machine that has no `libF77`. When you ship your program to your customer, you can ship `libF77` or you can link it into your program statically.

Example: Link `libF77` statically and link `libc` dynamically:

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

The order for the `-l`*x* options is important. Use the order shown in the example.

–nolibmil

No inline templates—reset `–fast` *not* to include inline templates.

Use this option *after* the `-fast` option, for example:

```
demo% f77 –fast –nolibmil …
```

–noqueue

License—do not queue a license request.

If you use this option, and no license is available, the compiler returns without queueing your request and without doing your compile. A nonzero status is returned for testing in `make` files.

-noreduction

iMPact—do no reduction with parallelization (*Solaris 2.x, SPARC only*).

This option requires the iMPact FORTRAN 77 multiprocessor enhancement package.

-norunpath

Library—put no run path in executable (*Solaris 2.x only).*

If an executable file uses shared libraries, then the compiler normally builds in a path that tells the runtime linker where to find those shared libraries. The path depends on the directory where you installed the compiler. 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 no wish to make the loader search down those paths when the executable is run at another site. Compare with `-R`*list.*

–N*xk*

Table sizes—reset internal compiler tables.

x must be one of `c`, `d`, `l`, `n`, `q`, `s`, or `x`.
k is an integer.

Make static tables in the compiler bigger. The compiler issues an error message if tables overflow, and suggests that you apply one or more of these flags. No spaces are allowed within this option string. The choices are:

`-Nc`*k*

Control statements. Maximum depth of nesting for control statements such as `DO` loops. The default is 25. Example: `f77 -Nc50 any.f`

`-Nd`*k*

Data structures. Maximum depth of nesting for data structures and unions. The default is 20. Example: `f77 -Nd30 any.f`

`-Nl`*k*

Continuation lines. Maximum number of continuation lines for a continued statement. The default is 99 (1 initial and 99 continuation). Any number greater than 19 is nonstandard. ♦ Example: `f77 -Nl200 any.f`

`-Nn`*k*

Identifiers. This option has no effect. The number of identifiers is now unlimited. This option is still recognized so it does not break make files, but it may be deleted in a future release.

`-Nq`*k*

Equivalence. Maximum number of equivalenced variables. The default is 500. Example: `f77 -Nq600 any.f`

`-Ns`*k*

Statement numbers. Maximum number of statement labels. The default is 2000. Example: `f77 -Ns3000 any.f`

`-Nx`*k*

External names. Maximum number of external names (common block names, subroutine and function names). The default is 1000. Example: `f77 -Nx2000 any.f`

–O[*n*] Performance—optimize for execution time.

n can be 1, 2, 3, 4, or 5. No space is allowed between `-O` and *n*. If `-O`[*n*] is not specified, the compiler still performs a default level of optimization; that is, it executes a single iteration of local common subexpression elimination and live/dead analysis.

`–g` does not suppress `–O`*n*, but `–O`*n* limits `–g` in certain ways; for details see *–g*, on page 52. For makefile changes regarding `-o` with `-g`, see *–g*, on page 52.

–O If you do not specify *n*, `f77` uses whatever *n* is most likely to yield the fastest performance for most reasonable applications. For the current release of FORTRAN 77, this is 3.

–O1 Do only the minimum amount of optimization (peephole). This is postpass assembly-level optimization. Do not use `-O1` unless `-O2` and `-O3` result in excessive compilation time, or running out of swap space.

–O2 Do basic local and global optimization. This level usually gives minimum code size. The details are: induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, and complex expression expansion.

Do not use `–O2` unless `–O3` results in excessive compilation time, running out of swap space, or excessively large code size.

–O3 Besides what `–O2` does, this option also optimizes references and definitions for external variables. Usually `–O3` makes larger code.

–O4 Besides what `–O3` does, this option also does automatic inlining of routines contained in the same file. It usually improves execution speed, but sometimes makes it worse. Usually `–O4` makes larger code.

For most programs, `–O4` is faster than `–O3` is faster than `–O2` is faster than `-O1`. But in a few cases, `–O2` might be better than the others, or `–O3` might be better than `–O4`. You can try compiling with each level to find if you have one of these rare cases.

The `–g` option suppresses the `–O4` automatic inlining described above.

The`-O3` and `-O4` options reduce the utility of debugging in that you cannot print variables from `dbx`, but you can use the `dbx` `where` command to get a symbolic traceback, without the penalty of turning off optimization.

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 command-line option. *(SPARC only*)

-O5 Attempt the highest level of optimization (*Solaris 2.x only*).

Use optimization algorithms that take more compilation time, or that do not have as high a certainty of improving execution time.

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

–o *nm* Output file—name the executable file *nm* instead of `a.out`.

There must be a blank between `-o` and *nm*.

-oldldo Output—use old list-directed output.

Omit the blank that starts each record for list-directed output. This is a change from releases 1.4 and earlier. The default behavior is to provide that blank, since the FORTRAN 77 Standard requires it. Note also the `FORM='PRINT'` option of `OPEN`. You can compile some source files with `-oldldo` and some without, in the same program.

–onetrip `DO` loops—use one trip `DO` loops.

Compile `DO` loops so that they are performed at least once if reached. `DO` loops in this FORTRAN 77 are not performed at all if the upper limit is smaller than the lower limit, unlike some implementations of FORTRAN 66 `DO` loops.

–p Profile by procedure for `prof`.

Prepare object files for profiling, see `prof` (1). If you compile and link in separate steps, and if you 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` with `gprof` does more.

–pad[=*p*] Insert padding for efficient use of cache.

This option inserts padding between arrays or character strings if they are:

- Static local and not initialized, or
- In common blocks

For either one, the arrays or character strings can not be equivalenced.

For `-pad[=p]`, if *p* is present, it must be one of the following:

`local`: Put padding between adjacent *local* variables.
`common`: Put padding between variables in common blocks.
`local,common`: Both local and common padding
`common,local`: Both local and common padding

Each `-pad` option string is *one* token—no internal spaces.

Defaults for `-pad`:

- Without the `-pad[=p]` option, `f77` does no padding.
- With `-pad`, but without the `=p`, `f77` does both local and common padding.

The following are equivalent:

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

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

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

For a definition of *local* variables, see *–stackvar*.

Restrictions on `-pad=common`

- Neither the arrays nor the character strings are equivalenced
- If `-pad=common` is specified for compiling a file that references a common block, it must be specified when compiling all files that reference that common block. The option changes the spacing of variables within the common block. If one program unit is compiled with the option and another is not, references to what should be the same location within the common block might reference different locations.
- If `-pad=common` is specified, the declarations of common block variables in different program units must be the same except for the names of the variables.The amount of padding inserted between variables in a common block depends on the declarations of those variables. If the variables differ in size or rank in different program units, even within the same file, the locations of the variables might not be the same.

- If `-pad=common` is specified, `EQUIVALENCE` declarations involving common block variables cause a fatal compilation error.

-parallel iMPact—Parallelize with: `-autopar`, `-explicitpar`, `-depend` (*SPARC, 2.x*).

Parallelize loops both automatically by the compiler and explicitly specified by the programmer. With explicit parallelization of loops, there is a risk of producing incorrect results. If optimization is not at `-O3` or higher, then it is raised to `-O3`.

The `-parallel` option reduces the utility of debugging (`-g`) in that you cannot print variables from `dbx`, but you can still use the `dbx where` command to get a symbolic traceback.

Avoid `-parallel` if you do your own thread management. See *–mt*.

The `-parallel` option requires the iMPact FORTRAN 77 multiprocessor enhancement package. To get faster code, use this option on a multiprocessor SPARC system. On a single-processor system, the generated code usually runs more slowly.

See Appendix C, "iMPact: Multiple Processors."

-pentium Generate code for Pentium (*x86 only*).

Generate code that exploits features available on x86 Pentium compatible computers. The default is `-386`. Code compiled with `-pentium` does run on 80386 and 80486 hardware, but it may be slower.

–pg Profile by procedure for `gprof`.

Produce counting code in the manner of `–p`, but invoke a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. Then you can make an execution profile by running `gprof` (1). `-pg` and `gprof` are complementary to `-a` and `tcov`.

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

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

For Solaris 2.x, when the operating system is installed, `gprof` is included if you do a developer install, rather than an end user install; it is also included if you install the package `SUNWbtool`.

–pic Library—produce position-independent code for shared library.

This kind of code is for dynamic shared libraries. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in program-counter-relative addressing mode through a procedure linkage table.

With `-pic` *(SPARC only*):

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

–PIC Library—similar to `-pic`, but with 32-bit addresses (*SPARC only*).

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

–Qoption *pr ls* Option—pass option list to the program *pr*.

Pass the option list *ls* to the program *pr*. There must be a blank between `Qoption` and *pr* and *ls*. The `Q` can be uppercase or lowercase. The list is a comma-delimited list of options, with no blanks within the list. Each option must be appropriate to that program, and can begin with a minus sign.

The program can be one of the following: `as`, `cg`, `cpp`, `fbe`, `f77pass0`, `f77pass1`, `iropt`, `ld`, or `ratfor`.

In Solaris 2.x, the assembler used by the compiler is named `fbe`.
In Solaris 1.x, it is called `as`.

Example: Pass the linker option `-s` to the linker `ld`:

```
demo% f77 –Qoption ld -s src.f
```

Example: Load map, 2.x:

Solaris 2.x

```
demo% f77 –Qoption ld -m src.f
```

Example: Load map, 1.x:

Solaris 1.x

```
demo% f77 –Qoption ld -M src.f
```

–R *ls* Library—store library paths in executable (*Solaris 2.x only*).

With this option, the linker, `ld`(1), stores a list of 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 your users run your executables without a special path option to find your dynamic libraries.

For `f77`, `-R` and the environment variable `LD_RUN_PATH` are *not* identical. They are identical, however, for the runtime linker, `ld.so`.

If you build `a.out` with:

- `-R`, then only the paths of `-R` are put in `a.out`. So `-R` is *raw*: it inserts only the paths you name, and no others.
- `LD_RUN_PATH`, then the paths of `LD_RUN_PATH` are put in `a.out`, plus paths for FORTRAN 77 libraries. So `LD_RUN_PATH` is *augmented*: it inserts the ones you name, plus various others.

- Both `LD_RUN_PATH` and `-R`, then only the paths of `-R` are put in `a.out`, and those of `LD_RUN_PATH` are *ignored*.

–r8 Set 8 byte default for `REAL,INTEGER,` and `LOGICAL`.

This option sets the default size for `REAL`, `INTEGER,` and `LOGICAL` to 8, and for `COMPLEX` to 16. *For `INTEGER` and `LOGICAL`, the compiler allocates 8 bytes, but does 4-byte arithmetic*. For *SPARC*, it sets the default size for `DOUBLE PRECISION` to 16, and for `DOUBLE COMPLEX` to 32.

`REAL` will be interpreted as `DOUBLE PRECISION`, `COMPLEX` as `DOUBLE COMPLEX`. For *SPARC*, `DOUBLE PRECISION` will be interpreted as quadruple precision and `DOUBLE COMPLEX` as quadruple complex.

This option applies to variables, literal constants, and intrinsic functions declared without an explicit byte size. As an intrinsic function example, `SQRT` is treated as `DSQRT`.

If you specify the size, then the default size is not used. For example, with `REAL*`*n* `R`, `INTEGER*`*n* `I`, `LOGICAL*`*n* `L`, and `COMPLEX*`*n* `Z`, the sizes of `R`, `I`, `L`, and `Z` are not affected by `-r8`.

In general, if you compile *one* subprogram with `-r8`, then be sure to compile *all* subprograms of that program with `-r8`. Similarly, for programs communicating through files in unformatted I/O, if one program is compiled with `-r8`, then the other program must also be compiled with `-r8`.

The impact on runtime performance may be great. With `-r8`, an expression like `float = 15.0d0*float` is evaluated in quadruple precision due to the declaration of the constant 15.

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

See also `-dble`.

-reduction iMPact—do reduction loops (*Solaris 2.x, SPARC only*).

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

The `-reduction` option requires the iMPact FORTRAN 77 multiprocessor enhancement package. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs more slowly.

See Appendix C, “iMPact: Multiple Processors."

Reduction works only during automatic parallelization. If you specify `-reduction` without `-autopar`, the compiler does no reduction. If you have a directive that explicitly specifies a loop, then there us no reduction for that loop.

Example: Automatically parallelize with *reduction*:

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

–S Assembly source—generate and leave only assembly source code.

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

–s Symbol table—strip the executable file of its symbol table.

This option makes the executable file smaller and more difficult to reverse engineer. However, this option prevents debugging.

–sb SourceBrowser—produce table information for the SourceBrowser.

–sbfast Similar to `-sb`, but faster, and makes no object files.

Produce *only* table information for SourceBrowser and stop. Do not assemble, link, or make object files.

–silent Show prompt only, reduce number of compiler messages.

Use this option to reduce the number of messages from the compiler. If there are no compilation warnings or errors, then show only the prompt. The default is to show the entry names and the file names.

–stackvar Allocate local variables on the stack for better optimizing with parallelization.

Use the stack to allocate all *local* variables and arrays in a routine unless otherwise specified. This option makes them automatic, rather than static, and provides more freedom to the optimizer for parallelizing a `CALL` in a loop.

Variables and arrays are local, unless they are:

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

You can get a segmentation fault using `-stackvar` with *large* arrays. Putting large arrays onto the stack can overflow the stack, so you may need to increase the stack size.

There are two stacks:

- The whole program has a *main* stack.
- Each thread of a multi-threaded program has a *thread* stack.

The default stack size is about 8 Megabytes for the main stack and 256 KBytes for each thread stack. The `limit` command (with no parameters) shows the current main stack size. If you get a segmentation fault using `-stackvar`, you might try doubling the main stack size at least once.

Example: Show the current *main* stack size:

The main stack size →

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

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

```
demo% limit stacksize 65536
```

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

```
demo% setenv STACKSIZE 8192
```

See `csh`(1) for details on the `limit` command.

–temp=*dir* Temporary files—define directory for temporary files.

Set directory for temporary files used by `f77` to be *dir*. No space is allowed within this option string. Without this option, the files are placed in: `/tmp/`.

–time Time for execution—show for each compilation pass.

–U Uppercase identifiers—leave identifiers in the original case.

Do not convert uppercase letters to lowercase, but leave them in the original case. The default is to convert to lowercase except within character-string constants.

If you debug FORTRAN 77 programs that use other languages, it is generally safer to compile with the `-U` option to get case-sensitive variable recognition. If you are not consistent in the case of your variables, the `-U` option can cause problems. That is, if you sometimes type `Delta`, `DELTA` or `delta`, then `-U` makes them different symbols.

–u Report undeclared variables.

Make the default type for all variables be *undeclared* rather than using FORTRAN 77 implicit typing. This option warns of undeclared variables, and does not override any `IMPLICIT` statements or explicit *type* statements.

-unroll=*n* Performance—unroll loops: direct the optimizer on unrolling loops.

n is a positive integer. The choices are:

- If *n*=1, this option *directs* the optimizer to unroll *no* loops (command).
- If *n*>1, this option *suggests* to the optimizer that it unroll loops *n* times.

If any loops are actually unrolled, then the executable file is larger.

-V Version ID—similar to `-v`, but also show version ID.

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

-v Show name of each compiler pass.

Show the name of each pass as the compiler executes, plus show in detail the options and environment variables used by the driver.

-vax=*v* Specify some coice of VMS features to use.

v must be one of the following: `align`, `misalign`, or `no`.

The definitions are:

- `-vax=align`

  Retain the *old* (release 3.0 and earlier) `-xl` behavior; that is, structures are *not* padded. If your program has misaligned structures, it will not run.

- `-vax=misalign`

  Same as `-vax=align`, except that this option, a synonym for `-xl`, allows structures to be misaligned.

- `-vax=no`

  Equivalent to not specifying `-xl` or `-vax=misalign`.

For details, see the description for `-xl`.

-vpara iMPact—show verbose parallelization warnings (*Solaris 2.x, SPARC only*).

As the compiler detects each explicitly parallelized loop that has dependencies, it issues a warning message, but the loop is still parallelized.

The `-vpara` option requires the iMPact FORTRAN 77 multiprocessor package.

Use `-vpara` with the `-explicitpar` option and the `C$PAR DOALL` directive.

Example: `-vpara` for verbose parallelization warnings:

```
demo% f77 -explicitpar -vpara any.f
```

-w Warnings—do not show warnings.

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

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

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

-xa Synonym for `-a`.

-xarch=*a* Limit the instructions `f77` can use (*SPARC, Solaris 2.x only*).

a must be one of: `generic`, `v7`, `v8a`, `v8`, `v8plus`, `v8plusa`.

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its primary use is to override a value supplied by the `-xtarget` option.

This option limits the instructions generated to those of the specified architecture, and *allows* the specified set of instructions. It does not guarantee an instruction is used; however, under optimization, it is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

`v7, v8`, and `v8a` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other and forward, but not backward.

For any particular choice, the generated executable may run much more slowly on earlier architectures.

`generic`: Get good performance on most SPARC systems.

This is the default. This option uses the best instruction set for good performance on most SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.

`v7`: Limit the instruction set to V7 architecture.

This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions.

This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions:

- The quad-precision floating-point instructions
- The integer `mul` and `div` instructions
- The `fsmuld` instruction

Examples: SPARCstation 1, SPARCstation 2

`v8a`: Limit the instruction set to the V8a version of the V8 architecture.

By definition, V8a means the V8 architecture, but without:

- The quad-precision floating-point instructions
- The `fsmuld` instruction

This option uses the best instruction set for good performance on the V8a architecture.

Example: Any machine based on the MicroSPARC I chip architecture

`v8`: Limit the instruction set to V8 architecture.

This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions.

Example: SPARCstation 10

`v8plus`: Limit the instruction set to the V8plus version of the V9 architecture.

By definition, V8plus means the V9 architecture, except:

- Without the quad-precision floating-point instructions
- Limited to the 32-bit subset defined by the V8plus specification
- Without the VIS instructions

This option uses the best instruction set for good performance on the V8plus chip architecture. In V8plus, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the `i`*x* and `l`*x* registers must not affect program results.

Example: Any machine based on the UltraSPARC chip architecture

Use of `-xarch=v8plus` causes the `.o` file to be marked as a V8+ binary. Such binaries will not run on a V7 or V8 machine.

`v8plusa`: Limit the instruction set to the V8plusa architecture variation.

By definition, V8plusa, means the V8plus architecture, plus:

- The UltraSPARC-specific instructions
- The VIS instructions

This option uses the best instruction set for good performance on the UltraSPARC™ architecture, but limited to the 32-bit subset defined by the V8plus specification.

Example: Any machine based on the UltraSPARC chip architecture

Use of `-xarch=v8plusa` also causes the `.o` file to be marked as a Sun-specific V8plus binary. Such binaries will not run on a V7 or V8 machine.

-xautopar Synonym for `-autopar` *(Solaris 2.x only)*.

-xcache=*c* Define cache properties for the optimizer (*SPARC, Solaris 2.x only*).

c must be one of the following:

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

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

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

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

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

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

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its primary use is to override a value supplied by the `-xtarget` option.

Table 2-15 `-xcache` Values

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

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

Level 1 cache has:
16K bytes
32 bytes line size
4-way associativity

Level 2 cache has:
1024K bytes
32 bytes line size
Direct mapping associativity

-xcg89 Synonym for `-cg89`.

-xcg92 Synonym for `-cg92`.

-xchip=*c* Specify processor for the optimizer (*SPARC, Solaris 2.x only*).

c must be one of: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`, `ultra`

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; its primary use is to override a value supplied by the `-xtarget` option.

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

- The ordering of instructions, that is, scheduling
- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

Table 2-16 `-xchip` Values

Value	Meaning
`generic`	Use timing properties for good performance on most SPARCs. This is the default value that directs the compiler to use the best timing properties for good performance on most SPARC processors, without major performance degradation on any of them.
`old`	Use timing properties of pre-SuperSPARC™ processors.
`super`	Use timing properties of the SuperSPARC chip.
`super2`	Use timing properties of the SuperSPARC II chip.
`micro`	Use timing properties of the MicroSPARC™ chip.
`micro2`	Use timing properties of the MicroSPARC II chip.
`hyper`	Use timing properties of the HyperSPARC™ chip.
`hyper2`	Use timing properties of the HyperSPARC II chip.
`powerup`	Use timing properties of the Weitek® PowerUp™ chip.
`ultra`	Use timing properties of the UltraSPARC chip.

`-xdepend` Synonym for `-depend` *(Solaris 2.x only).*

`-xexplicitpar` Synonym for `-explicitpar` *(Solaris 2.x only).*

-xF Function reorder—allow function-level reordering (*Solaris 2.x only*).

Allow the reordering of functions in the core image using the compiler, the Analyzer and the linker. If you compile with the `-xF` option, then run the Analyzer, you get a map file that shows an optimized order for the functions. The 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. Within the mapfile, if you include the flag `O` (that's an *oh*, for *order*, not a *zero*) in the string of segment flags, then the static linker `ld` attempts to place sections in the order they appear in the mapfile.

Example: In the mapfile, there can be a line such as:

```
text = LOAD ? RXO
```

See the `analyzer`(1) and `debugger`(1) `man` pages.

-xhelp=*h* Show help information for `README` file or options (flags).

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

`readme`: Show the online `README` file.
`flags`: Show the compiler flags (options).

`-xhelp=flags` is a synonym for `-help`.

-xildoff Turn off the Incremental Linker. (*SPARC, Solaris 2.x only*).

This forces the use of the standard linker, `ld`.

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

Override this default by using the `-xildon` option.

-xildon Turn on the Incremental Linker (*SPARC, Solaris 2.x only*).

Turn on the Incremental Linker and force the use of `ild` in incremental mode.

This option is the default if you use `-g` and do *not* use `-G`, and do *not* name any source file on the command line.

Override this default by using the `-xildoff` option.

`-xinline=`*rlst* Synonym for `-inline=`*rlst*.

`-xl[d]` Extend the language with more VMS FORTRAN features.

`-xl`: Extend the language with more VMS features. This is a macro that is translated to `-vax=misalign`, and provides the language features that are listed later in this description.

Although you get most VMS features automatically, without any special options, you must use the `-xl` option for a few VMS features.

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

The following VMS language features require the `-xl[d]` option:

- Unformatted record size in words rather than bytes (`-xl`)
- VMS style logical file names (`-xl`)
- Quote (") character introducing octal constants (`-xl`)
- Backslash (\) as ordinary character within character constants (`-xl`)
- Nonstandard form of the `PARAMETER` statement (`-xl`)
- Alignment of structures as in VMS. (`-xl`)
- Debugging lines as comment lines or FORTRAN 77 statements (`-xld`)

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

If you use both `-oldstruct` and `-xl`, then you get `-oldstruct`. If you need to share structures with C, use the default; do not use `-xl` and do not use `-oldstruct`.

You may also be interested in `-lV77` and the VMS library. See “Libraries Provided with the Compiler” on page 168.

Read the chapter on VMS language extensions in the *FORTRAN 77 4.0 Reference Manual* for details of the VMS features that you get automatically.

-xld VMS—Debug comments: extended language, VMS, plus *debug* comments.

In addition to the features you get with `-xl`, the `-xld` option causes debugging comments (`D` or `d` in column one) to be compiled. Without the `-xld` option, they remain comments only. No space is allowed between `-xl` and `d`.

-xlibmil Synonym for `-libmil`.

-xlibmopt Use library of optimized math routines (*SPARC only*).

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.

-xlicinfo License information—show license server user IDs.

Use this option to return license information about the licensing system—in particular, the name of the license server and the user ID for each of the users who have licenses checked out.

Generally, with this option, no compilation takes place, and a license is not checked out. Also, this option is normally used with no other options. However, if a conflicting option is used, then the last one on the command line prevails, and there is a warning.

Example: Report license information, do not compile; the order counts:

```
demo% f77  -c -xlicinfo any.f
```

Example: Do not report license information, do compile; the order counts:

-Xlist Do global program checking.

This option helps find a variety of bugs by checking across routines for consistency in arguments, common blocks, parameters, and so forth. In general, `-Xlist` also makes a line-numbered listing of the source and a cross reference table of the identifiers. The errors that are found do not prevent the program from being compiled and linked.

Example: Check across routines for consistency:

```
demo% f77 -Xlist  fil.f
```

The above example shows the following in 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)

See "Global Program Checking (-Xlist)" on page 173," for details.

-xloopinfo Synonym for `-loopinfo` *(Solaris 2.x only)*.

-xnolib Synonym for `-nolib`.

-xnolibmil Synonym for `-nolibmil`.

-xnolibmopt Do not use fast math library *(SPARC only)*.

Reset `-fast` so that it does not use the library of selected math routines optimized for performance. Use this after the `-fast` option:
`f77 -fast -xnolibmopt` …

-xO[*n*] Synonym for `-O`[*n*].

-xparallel Synonym for `-parallel` *(Solaris 2.x only)*.

-xpg Synonym for `-pg`.

-xprofile=*p* Collect or use data for profile to optimize (*SPARC, Solaris 2.x only*).

p must be one of `collect`, `use[:`*nm*`]`, or `tcov`.

`collect`

Collect and save execution frequency data for later use by optimizer via `-xprofile=use` to improve optimization at a later compilation of the program.

`f77` compiles code to measure execution frequency at a low level. During execution, execution frequency data is written to the file *binary_name*`.prof/feedback`. If you run the program several times, the execution frequency data accumulates in the `feedback` file; that is, output from prior runs is not lost.

`use[:`*nm*`]`

Use execution frequency data to optimize strategically.

The *nm* is the name of the executable that is being analyzed. This name is optional. If *nm* is not specified, `a.out` is assumed to be the name of the executable.

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

The source files and the compiler options (excepting only this option), must be exactly the same as for the compilation used to create the compiled program that was executed to create the `feedback` file.

`tcov`

Collect data for programs with source code in header files.

This option is also good for programs which use C++ templates. Header files or C++ templates are very unusual for Fortran programs, so most Fortran users can safely ignore the `tcov` value for `-xprofile`.

Code instrumentation is similar to that of `-a`, but `.d` files are no longer generated. Instead, a single file is generated, whose name is based on the name of the final executable. For example, if `/xy/stuff` is the executable file, then `/xy/stuff.profile/prog.tcovd` is the data file.

When running `tcov`, you must pass it the `-x` option to make it use the new style of data. If not, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output.

Unlike `-a`, the `TCOVDIR` environment variable has no effect at compile-time. However, its value is used at program runtime.

See `-a` for information on the old style of profiling; see also the `tcov`(1) man page, and the *Profiling Tools* manual for more details.

-xreduction

Synonym for `-reduction` *(Solaris 2.x only)*.

-xregs=*r*

Specify register usage (*SPARC, Solaris 2.x only*).

r is a comma-separated list that consists of one or more of the following:

[no%]`appl`, [no%]`float`.

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

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

`appl`: Allow using the registers `g2`, `g3`, and `g4`.

In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.

`no%appl`: Do not use the `appl` registers.

`float`: Allow using the floating-point registers as specified in the SPARC ABI.

You can use these registers even if the program contains no floating-point code.

`no%float`: Do not use the floating-point registers.

With this option, a source program cannot contain any floating-point code.

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

-xs Allow debugging by `dbx` without object (`.o`) files (*Solaris 2.x only*).

With `-xs`, if you move executables to another directory, then you can use `dbx` and ignore the object (`.o`) files. Use this option in case you cannot keep the `.o` files around.

- `f77` passes `-s` to the assembler and then the linker places all symbol tables for `dbx` in the executable file.
- This way of handling symbol tables is the older way. It is sometimes called *no auto-read*.
- The linker links more slowly, and `dbx` initializes more slowly.

Without `-xs`, if you move the executables, you must move both the source files and the object (`.o`) files, or set the path with either the dbx `pathmap` or `use` command.

- This way of handling symbol tables is the newer and default way of loading symbol tables. It is sometimes called *auto-read*.
- The symbol tables are distributed in the `.o` files so that `dbx` loads the symbol table information only if and when it is needed. Hence, the linker links faster, and `dbx` initializes faster.

-xsafe=mem Assume no memory-based traps (*SPARC, Solaris 2.x only*).

This allows `f77` to assume no memory-based traps occur. It grants permission to use the speculative load instruction on V9 machines.

-xsb Synonym for `-sb`.

-xsbfast Synonym for `-sbfast`.

-xspace Do not increase code size (*SPARC, Solaris 2.x only*).

Do no optimizations that increase the code size.

Example: Do not unroll loops.

-xtarget=*t* Specify system for optimization (*SPARC, Solaris 2.x only*).

Specify the target system for the instruction set and optimization.

t must be one of: `native`, `generic`, *system-name*.

The `-xtarget` option permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on real systems. The only meaning of `-xtarget` is in its expansion.

The performance of some programs may benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a `generic` specification is sufficient.

`native`: Optimize performance for the host system.

The compiler generates code optimized for the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.

`generic`: Get the best performance for generic architecture, chip, and cache.

The compiler expands `-xtarget=generic` to:

```
-xarch=generic -xchip=generic -xcache=generic
```

This is the default value.

system-name: Get the best performance for the specified system.

You select a system name from Table 2-17 that lists the mnemonic encodings of the actual system names and numbers.

This option is a macro. Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options, as shown in Table 2-17. `fpversion`(1) can be run to determine the target definitions on any system.

For example:

`-xtarget=sun4/15` means `-xarch=v8a -xchip=micro -xcache=2/16/1`

Table 2-17 `-xtarget` Expansions

-xtarget	-xarch	-xchip	-xcache
sun4/15	v8a	micro	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/30	v8a	micro	2/16/1
sun4/40	v7	old	64/16/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/65	v7	old	64/16/1
sun4/75	v7	old	64/32/1
sun4/110	v7	old	2/16/1
sun4/150	v7	old	2/16/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/630	v7	old	64/32/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1

Table 2-17 `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/64/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss2p	v7	powerup	664/32/1
ss4	v8a	micros2	8/64/1
ss5	v8a	micro2	8/64/1
ssvyger	v8a	micro2	8/64/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1

Table 2-17 `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ss10/514	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss20/51	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/502	v8	super	16/32/4
ss10/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
ss1000	v8	super	16/32/4:1024/32/1

Table 2-17 `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
sc2000	v8	super	16/32/4:1024/64/1
cs6400	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/1170	v8	ultra	16/32/1:512/64/1
ultra1/2170	v8	ultra	16/32/1:512/64/1
ultra1/2200	v8	ultra	16/32/1:1024/64/1

-xtime

Synonym for `-time`.

-xunroll=*n*

Synonym for `-unroll=`*n*.

-xvpara

Synonym for `-vpara` *(Solaris 2.x only)*.

-Zlp

iMPact—prepare for profiling by `looptool` (*Solaris 2.x, SPARC only*).

Prepare object files for the loop profiler, `looptool`. The `looptool`(1) utility can then be run to generate loop statistics about the program.

The `-Zlp` option requires the iMPact FORTRAN 77 multiprocessor package.

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

If you compile *one* subprogram with `-Zlp`, you need not compile *all* the subprograms of that program with `-Zlp`. However, you receive the loop information only for the files compiled with `-Zlp`, and no indication that the program includes other files.

Refer to the *Thread Analyzer User's Guide* for more information.

-ztext Library—make no library with relocations *(Solaris 2.x only)*.

Do not make the library if relocations remain. The general purpose of `-ztext` is to ask if the generated library is pure text; instructions are all position-independent code. Therefore, it is generally used with both `-G` and `-pic`.

With `-ztext`, if `ld` finds an incomplete relocation in the *text* segment, then it does not build the library. If it finds one in the *data* segment, then it generally builds the library anyway; the data segment is writable.

Without `-ztext`, `ld` builds the library, relocations or not.

A typical use is to make a library from both source files and object files, where you do not know if the object files were made with `-pic`.

Example: Make library from both source and object files:

```
demo% f77 -G -pic -ztext -o MyLib -hMyLib a.f b.f x.o y.o
```

An alternate use is to ask if the code is position-independent already: compile without `-pic`, but ask if it is pure text.

Example: Ask if it is pure text already—even without `-pic`:

```
demo% f77 -G -ztext -o MyLib -hMyLib a.f b.f x.o y.o
```

If you compile with `-ztext` and `ld` does not build the library, then you can recompile without `-ztext`, and `ld` will build the library. The failure to build with `-ztext` means that one or more components of the library cannot be shared; however, maybe some of the other components can be shared. This raises questions of performance that are best left to you, the programmer.

-Ztha iMPact— prepare for Thread Analyzer (*Solaris 2.x, SPARC only*).

Prepare object files for Thread Analyzer. This option inserts calls to a profiling library at all procedure entries and exits. Code compiled with `-Ztha` links with the library `libtha.so`. The `-Ztha` option requires the iMPact FORTRAN 77 MP package.

If you compile and link in separate steps, and you compile with `-Ztha`, then link with `-Ztha`.

If you compile a subprogram with `-Ztha`, you need not compile all subprograms of that program with `-Ztha`. However, you get thread statistics only for the files compiled with `-Ztha`, and no indication that the program includes other files.

Refer to `tha` (1) or the *Thread Analyzer User's Guide* for more information.

2.9 Directives

A directive passes information to a compiler in a special form of comment. ♦

Compiler directives are also called *pragmas*. There are two kinds of directives:

- General directives
- Parallel directives

General Directives

The form of a general directive is one of the following:

- `C$PRAGMA` *id*
- `C$PRAGMA` *id* (*a* [, *a*] …) [, *id* (*a* [, *a*] …)] ,…
- `C$PRAGMA SUN` *id*

The variable *id* identifies the specific directive; *a* is an argument.

Syntax

A general directive has the following syntax:

- In column one, any of the comment-indicator characters `c`, `C`, `!`, or `*`
- The next 7 characters are `$PRAGMA`, no blanks, any uppercase or lowercase
- In *any* column, the `!` comment-indicator character

Rules and Restrictions

After the first eight characters, blanks are ignored, and uppercase and lowercase are equivalent, as in FORTRAN 77 text.

Because it is a comment, a directive cannot be continued, but you can have many C$PRAGMA lines, one after the other, as needed.

If a comment satisfies the above syntax, it is expected to contain one or more directives recognized by the compiler; if it does not, a warning is issued.

The C Directive

The C() directive specifies that its arguments are external functions written in the C language. It is equivalent to an EXTERNAL declaration with the addition that the FORTRAN 77 compiler does not append an underscore to such names, as it ordinarily does with external names. See Chapter 12, "C–FORTRAN 77 Interface," for more details.

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

Example: To compile ABC and XYZ for C:

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

The UNROLL Directive

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

The C$PRAGMA SUN UNROLL=*n* directive instructs the optimizer to unroll loops *n* times.

n is a positive integer. The choices are:

- If *n*=1, this directive directs the optimizer *not* to unroll any loops.
- If *n*>1, this directive suggests to the optimizer that it unroll loops *n* times.

If any loops are actually unrolled, then the executable file becomes larger.

Example: To unroll loops two times:

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

Parallel Directives

A *parallel* directive directs the compiler to do some parallelizing. The syntax of parallel directives is different from the syntax of general directives.

Syntax

A parallel directive has the following syntax:

- The first character must be in column one.
- The first character can be any one of `c`, `C`, `*`, or `!`.
- The next 4 characters are `$PAR`, no blanks, any uppercase and lowercase.

A parallel directive differs slightly from the more general directive in the following ways:

- A parallel directive must start in column one.
- The initial characters are `C$PAR`, `*$PAR`, `c$par`, `*$par`,…

Usage

Currently, there are three parallel directives for explicit parallelization:

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

See Appendix C, “iMPact: Multiple Processors.”

2.10 Native Language Support

This version of FORTRAN 77 supports the development of applications using languages other than English, including most European languages. As a result, you can switch the development of applications from one native language to another.

This FORTRAN 77 compiler implements internationalization as follows:

- It recognizes 8-bit characters from European keyboards supported by Sun.
- It is 8-bit clean and allows the printing of your own messages in the native language.
- It allows native language characters in comments, strings, and data.
- It allows you to localize the compile-time error messages files.

Locale

You can enable changing your application from one native language to another by setting the locale. Doing so changes some aspects of displays, such as date and time formats.

For information on this and other native language support features, read Chapter 6, “Native Language Application Support,” of the *System Services Overview* for Solaris software.

Even though some aspects can change if you set the locale, certain aspects cannot change. An internationalized compiler language does not allow input and output in the various international formats. If it does, it does not comply with the language standard appropriate for its language. For example, some languages have standards that specify a period (.) as the decimal unit in the floating-point representation.

Example: No I/O in international formats:

native.f

```
      PROGRAM sample
      REAL r
      r = 1.2
      WRITE( 6, 1 ) r
   1  FORMAT( 1X F10.5 )
      END
```

Here is the output:

```
       1.20000
```

In the example above, if you reset your system locale to, say, France, and rerun the program, you still receive the same output. The period is not replaced with a comma, the French decimal unit.

Compile-Time Error Messages

The compile-time error messages are on files called source catalogs so you can edit them. You may decide to translate them to a local language such as French or Italian. Usually, a third party does the translating. Then you can make the results available to all local users of `f77`. Each user of `f77` can choose to use these files or not.

Localizing and Installing the Files

Usually a system administrator does the installation. It is generally done only once per language for the whole system, rather than once for each user of `f77`. The results are available to all users of `f77` on the system.

1. **Find the source catalogs.**
   The file names are:
   - `SUNWspro_f77pass1_srccat` (about 300 error messages)
   - `SUNWspro_compile_srccat` (about 10 error messages)
2. **Edit the source catalogs.**
   a. **Make backup copies of the files.**
   b. **In whatever editor you are comfortable with, edit the files.**
   The editor can be `vi`, `emacs`, `textedit`, and so forth.
   Preserve any existing format directives, such as `%f`, `%d`, `%s`, and so forth.
   c. **Save the files.**
3. **Generate the binary catalogs from the source catalogs.**
   The compiler uses only binary catalogs. Run the `gencat` program twice.
   a. **Read the** `SUNWspro_f77pass1_srccat` **source file and generate the** `SUNWspro_f77pass1_cat` **binary file.**

```
demo% gencat SUNWspro_f77pass1_cat  SUNWspro_f77pass1_srccat
```

b. Read the `SUNWspro_compile_srccat` **source file and generate the** `SUNWspro_compile_cat` **binary file.**

```
demo% gencat SUNWspro_compile_cat  SUNWspro_compile_srccat
```

4. Make the binaries available to the general user.
Either put the binary catalogs into the standard location or put the path for them into the environment variable `NLSPATH`.

a. Define the standard location and name.
Put the files into the directory indicated:

Solaris 2.x `/opt/SUNWspro/lib/locale/`*lang*`/LC_MESSAGES/`

Solaris 1.x `/usr/share/lib/locale/`*lang*`/LC_MESSAGES/`

where *lang* is the directory for the particular (natural) language. For example, the value of *lang* for Italian is *it*.

b. Set up the environment variable.
Put the path for the new files into the `NLSPATH` environment variable. For example, if your files are in `/usr/local/MyMessDir/`, then use the following commands.

In a `sh` shell:

```
demo$ NLSPATH=/usr/local/MyMessDir
demo$ export NLSPATH
```

In a `csh` shell:

```
demo% setenv NLSPATH  /usr/local/MyMessDir
```

The `NLSPATH` variable is standard for the X/Open environment. For more information, read the X/Open documents.

Using the File After Installation

You use the file by setting the environment variable LC_MESSAGES. This setup is generally done once for each developer.

Example: Set the environment variable LC_MESSAGES:

This example assumes standard install locations, and that the messages are localized in Italian.

In a sh shell:

```
demo% LC_MESSAGES it
demo% export it
```

In a csh shell:

```
demo% setenv  LC_MESSAGES it
```

2.11 Miscellaneous Tips

Here are some suggestions on how to use the compiler.

Floating-Point Hardware Type

Some compiler options are specific to particular hardware options. The utility fpversion tells which floating-point hardware is installed. The utility fpversion(1) takes 30 to 60 wall clock seconds before it returns, since it dynamically calculates hardware clock rates of the CPU and FPU.

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

Many Options on Short Commands

You may use long command lines with many options. To simplify the task, make a special alias or use environment variables.

Alias Method

Example: Define `f77f`:

```
demo% alias f77f "f77 -silent -fast -O4"
```

Example: Use `f77f`:

```
demo% f77f any.f
```

`f77f` is now the same as: `f77 -silent -fast -O4 any.f`.

Environment Variable Method

You can shorten command lines by using environment variables. The `FFLAGS` or `OPTIONS` variables are special variables for FORTRAN.

- If you set `FFLAGS` or `OPTIONS`, they can be used in the command line.
- If you are compiling with `make` files, `FFLAGS` is used automatically if the `make` file uses only the implicit compilation rules.

Example: Set `FFLAGS`:

```
demo% setenv FFLAGS '-silent -fast -O4'
```

- Example: Use `FFLAGS` explicitly:

```
demo% f77 $FFLAGS any.f
```

  `f77 $FFLAGS` is now the same as: `f77 -silent -fast -O4 any.f`.

- Example: Let `make` use `FFLAGS` implicitly:

  If both:

  - The compile in a makefile is *implicit*, that is, *no* explicit `f77` compile line
  - The `FFLAGS` variable is set as above

Then invoking the make file results in a compile command equivalent to: f77 -silent -fast -O4 any.f.

Align Block

In Solaris 2.x, the –align _*block*_ option is available only for compatibility with old makefiles. It is recognized, so it does not break such files, but it does not perform any function. However, you can still page-align a common block.

This rule applies to *uninitialized* data only. If any variable of the common block is initialized in a DATA statement, then the block is not aligned. This aligns the common block on a page boundary. Its size is increased to a whole number of pages; its first byte is placed at the beginning of a page.

Example: Page-align the common block whose FORTRAN 77 name is *block*:

```
      COMMON /BLOCK/ A, B
      REAL*4 A(11284), B(11284)
      ...
```

This block has a size of 90,272 bytes. You must create a separate assembler source file (.s file) consisting of the following .common statement:

```
demo% cat comblk.s
.common block_,90272,4096
demo%
```

In this example:

- block is the f77 name of the block with an appended underscore (_).
- 90272 is the block size in bytes.
- 4096 is the page size in bytes. Some systems have different page sizes.

If you do *not* use the -U option, use lowercase for the common block name. If you *do* use -U, use the case of the common block name in your source code. The parameters are *block name*, *block size*, and *page size*.

You must compile and link the `.s` file with the `.f` files:

```
demo% f77 any.f comblk.s
```

The stricter alignment from this file should override the less strict alignment for the common block from the other `.o` files.

Optimizer Out of Memory

Optimizers use a lot of memory. For SPARC systems, 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.

It is recommended that you have at least 24 Megabytes of memory. If you do full optimization, at least 32 Megabytes are recommended. How much you need 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.

If the optimizer runs out of swap space, try any of the following measures, which are listed in increasing order of difficulty:

- Change from `–O3` to `–O2`.
- Use `fsplit` to divide multiple-routine files into files of one routine per file.
- Allow space for a bigger swap file. See `mkfile`(8).

  Example (*2.x*): Become superuser, make 90-Megabyte file, tell OS to use it:

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

  Example (*1.x*): Become superuser, make the file, and instruct OS to use it:

```
demo# mkfile -v 20m /home/swapfile
/home/swapfile 20971520 bytes
demo# swapon  /home/swapfile
```

The above swap command must be reissued every time you reboot, or added to the appropriate `/etc/rc` file. The Solaris 2.x command, `swap -s`, displays available swap space. See `swap`(1M).

Control of Virtual Memory

If you optimize at `-O3` or `-O4` with very large routines (thousands of lines of code in a single procedure), the optimizer may require an unreasonable amount of memory. In such cases, performance of the machine may be degraded. You can control this by limiting the amount of virtual memory available to a single process.

Virtual Memory Limits

To limit virtual memory:

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

  Example: Limit virtual memory to 16 Megabytes:

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

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

  Example: Limit virtual memory to 16 Megabytes:

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

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

This limit cannot be greater than the machine's total available swap space and, in practice, must be small enough to permit normal use of the machine while a large compilation is in progress.

Be sure that no compilation consumes more than half the space.

Example: With 32 Megabytes 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 of data size depends on the degree of optimization requested, and the amount of real memory and virtual memory available.

Swap Space Limits

You can determine the actual swap space from either sh or csh.

Example: Use the swap command in 2.x:

Solaris 2.x

```
demo% swap -s
```

Example: Use the pstat command in 1.x:

Solaris 1.x

```
demo% pstat -s
```

Memory

You can also determine the actual real memory from either sh or csh.

- Example: Use the following command in 2.x:

Solaris 2.x

```
demo% /usr/sbin/dmesg | grep mem
```

- Use either of the following commands in 1.x:

Solaris 1.x

```
demo% /etc/dmesg | grep mem
```

or:

Solaris 1.x

```
demo% grep mem /var/adm/messages*
```

BCP Mode: How to Make 1.x Applications Under 2.x

This section shows some details of how to, in a Solaris 2.x operating environment, compile and link applications that run in a Solaris 1.x operating environment.

Note – Even though it is possible, it is not recommended to produce 1.x executables on a 2.x development platform. Most developers consider it too complicated.

Read the SunSoft publication, *Solaris 2.3 Binary Compatibility Manual*, first.

The usual way is as follows:

- The 1.x compilers in `/usr/lang/` are used on 1.x platforms to produce 1.x executables.
- The 2.x compilers in `/opt/SUNWspro/bin` are used on 2.x platforms to produce 2.x executables.

To use a 2.x operating environment to make executables that run under 1.x, use the following steps:

- Be sure the appropriate BCP packages are installed:
  - `SUNWbcp`: Binary Compatibility
  - `SUNWscbcp`: SPARCompiler Binary Compatibility

  Use `pkginfo` to verify the installation. The binary compatibility libraries are installed in `/usr/4lib`.

- Be sure to use the 1.x compiler, not the native 2.x compiler.

  You may need to install the 1.x compiler in a nonstandard location to make it accessible on the 2.x platform.

- If possible, perform the final link of object files on a 1.x platform.

  Otherwise, to link your 1.x executable successfully on the 2.x platform, you need access to a 1.x version of `ld`. do the following:

  a. Copy a 1.x version of `ld` to *1.x_ld_path*/`ld`.

  b. Tell `f77` where to find the 1.x linker by supplying its path via `-Qpath`.

- Make versions of the 1.x libraries available on 2.x, as follows:

  On 1.x, copy files from `/lib` to *com_dir*:

```
demo% cp -p /lib/libc.a com_dir
demo% cp -p /lib/libc.so.1.8 com_dir
demo% cp -p /lib/libc.sa.1.8 com_dir
Plus any other system 1.x libraries you need
demo%
```

  On 2.x, break the link `/lib -> /usr/lib`:

```
demo% su root
Password: your_root_password
#mv /lib /lib-                    Rename the link /lib to /lib-
#mkdir /lib                       Make a new directory /lib
#demo%
```

  On 2.x, copy the same files from *com_dir* to `/lib`:

```
#mv com_dir/libc.a /lib Move the same files from com_dir to /lib
#mv com_dir/libc.so.1.8 /lib
#mv com_dir/libc.sa.1.8 /lib
Plus any other system 1.x libraries you need
#exit
demo%
```

Or, you can move the needed libraries into a directory of your choosing and enable the linker to find these libraries by supplying the path via the `-L` option or the `LD_LIBRARY_PATH` environment variable.

Make sure 1.x libraries that are non-system libraries are available, say, in the *1.x_lib_path*.

- Compile the program:

```
demo% 1.x_f77_path/f77 -Qpath 1.x_ld_path -L1.x_lib_path file.f
```

Where:
- `f77` is in *1.x_f77_path/*
- `ld` is in *1.x_ld_path*
- Libraries are in *1.x_lib_path*

Be aware of these pitfalls:

- If `libc` is linked statically then all libraries must be linked statically.
- `/usr/4lib` is not a suitable choice for *1.x_lib_path* or for system libraries you copy, because the presence of `libc.so.101.8` and `libc.so.102.8` frustrates linking.
- The message, `bad magic number`, probably means that you attempted to link with a 2.x library instead of a 1.x library. Check your command line for inappropriate `-L` options; also check `LD_LIBRARY_PATH`. Remember that the 1.x linker searches the following paths for libraries:

Environment variable	`LD_LIBRARY_PATH`
Directories specified at link-time	`-L`*dir*
Default directories	`/lib:/usr/lib:/usr/local/lib`

- Normally, `LD_LIBRARY_PATH` points to 2.x libraries on a 2.x platform; it may need to be reset, however. In particular, do not put `/usr/lib` in `LD_LIBRARY_PATH`.

Here is a summary:

- The resultant `a.out` should run on 1.x systems. It can run in BCP mode on 2.x systems, including the development platform. See the *Solaris 2.3 Binary Compatibility Manual* for guidelines.
- You may want to replace your 1.x shared libraries in 1.*x_lib_path* with 2.x libraries, since `a.out` tries to link with them.
- Use `ldd` to find which shared libraries `a.out` links with.
- On a 1.x system, `a.out` looks for shared libraries in the directories that were found on your development platform; differences in directory structure may cause problems.

File System and FORTRAN 77 I/O 3

This chapter is a basic introduction to the file system and how it relates to the FORTRAN 77 I/O system. If you understand these concepts, skip this chapter.

This chapter is organized into the following sections.

3.1 Summary

The basic file system consists of a hierarchical file structure, established rules for file names and path names, and various commands for moving around in the file system, showing your current location in the file system, and making, deleting, or moving files or directories.

The system file structure of the UNIX operating system is analogous to an upside-down tree. The top of the file system is the *root* directory. Directories, subdirectories, and files all branch down from the root. Directories and subdirectories are considered nodes on the directory tree, and can have subdirectories or ordinary files branching down from them. The only directory that is not a subdirectory is the root directory, so except for this instance, you do not usually make a distinction between directories and subdirectories.

A sequence of branching directory names and a file name in the file system tree describes a *path*. Files are at the ends of paths, and cannot have anything branching from them. When moving around in the file system, *down* means away from the root; *up* means toward the root. Figure 3-1 shows a diagram of a file system tree structure.

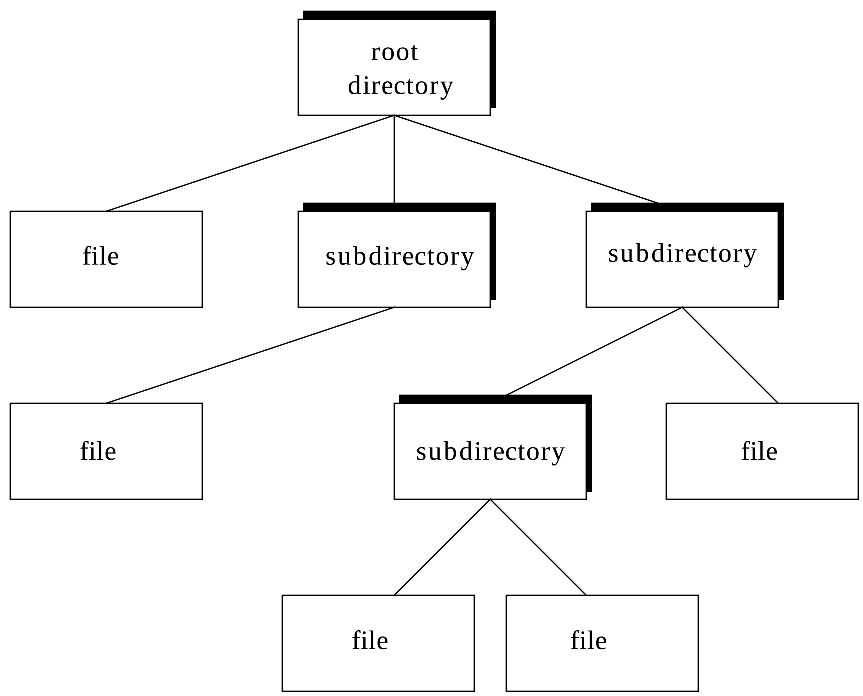


Figure 3-1 File System Hierarchy

3.2 Directories

All files branch from directories, except the root directory. Directories are just files with special properties. While you are logged on, you are *in a directory.*

When you first log on, you are usually in your *home* directory. At any time, wherever you are, the directory you are in is called your *current working directory*. It is often useful to list your current working directory. The `pwd` command prints the current working directory name; the `getcwd` routine returns the current working directory name.

You can change your current working directory simply by moving to another directory. The `cd` shell command and the `chdir` routine change the current working directory to a different directory.

3.3 File Names

All files have names, and you can use almost any character in a file name. The name can be up to 1,024 characters long, but individual components can be only 512 characters long.

To prevent the shell from misinterpreting certain special punctuation characters, restrict your use of punctuation in file names to the dot (`.`), underscore (_), comma (`,`), plus (+), and minus (–). The slash (`/`) character has a specific meaning in a file name, and is only used to separate components of the path name, as described in the following section. Also, avoid using blanks in file names. Directories are just files with special properties and follow the same naming rules as files. The only exception is the root directory, named slash (`/`).

3.4 Path Names

To describe a file anywhere in the file system, you can list the sequence of names for the directory, subdirectory, and so forth; and the file, separated by slash characters, down to the file you want to describe.

If you show *all* the directories, starting at the root, that is called an *absolute* path name. If you show only the directories below the current directory, that is called a *relative* path name.

Relative Path Names

From anywhere in the directory structure, you can describe a *relative path name* of a file. Relative path names start with the directory you are in—the current directory—instead of the root.

For example, if you are in the directory `/usr/you`, and you use the relative path name, `mail/record`, that is equivalent to using the absolute path name, `/usr/you/mail/record`.

See this illustration:

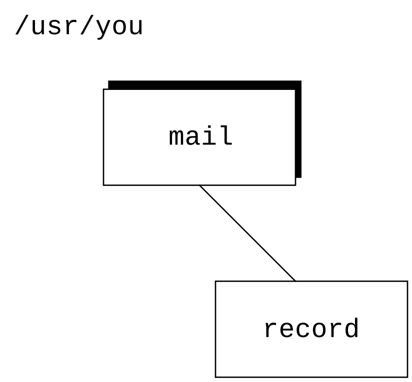


Figure 3-2 Relative Path Name

Absolute Path Names

A list of directories and a file name, separated by slash characters, from the root to the file you want to describe, is called an *absolute path name*. It is also called the *complete file specification* or the *complete path name*.

A complete file specification has the general form:

/*directory*/*directory*/*…*/*directory*/*file*

There can be any number of directory names between the root (/) and the file at the end of the path, as long as the total number of characters in a given path name is less than or equal to 1,024.

An absolute path name is illustrated in the following diagram:
`/usr/you/mail/record`

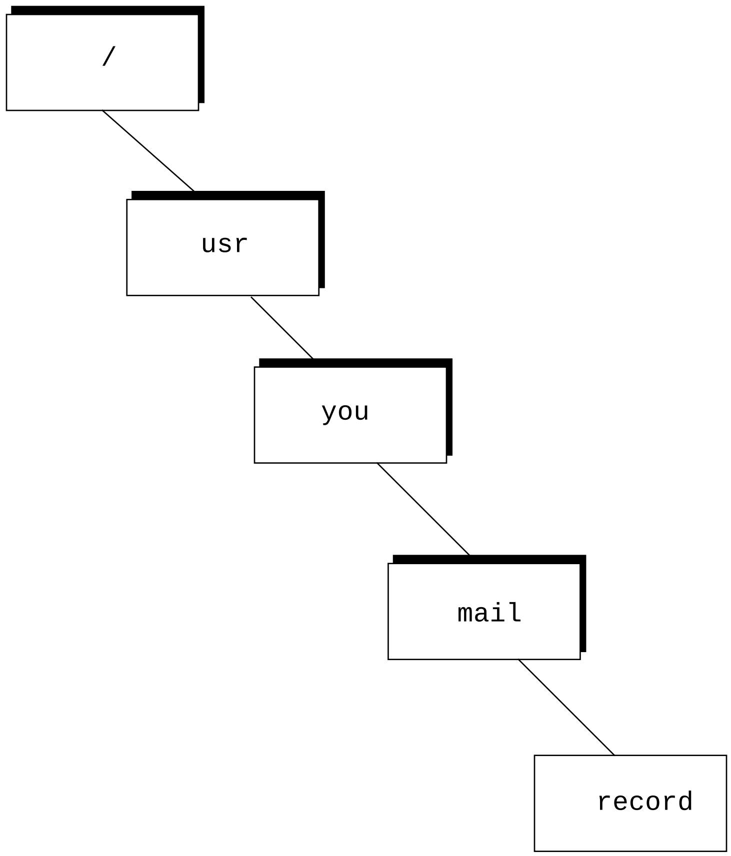


Figure 3-3 Absolute Path Name

3.5 Redirection

Redirection is a way of changing the files that a program uses without passing a file name to the program. Both input to and output from a program can be redirected.

The usual symbol for redirecting standard input is the < sign; for standard output, it is the > sign. File redirection is a function performed by the command interpreter or shell when a program is invoked by it.

Input

The shell command line for `myprog` to read from `mydata` is:

```
demo% myprog < mydata
```

The above command causes the file `mydata`, which must already exist, to be connected to the standard input of the program `myprog` when it is run. This means that if `myprog` is a FORTRAN 77 program and reads from unit 5, it reads from the `mydata` file.

Output/Truncate

The shell command line for `myprog` to *write* to `myoutput` is:

```
demo% myprog > myoutput
```

The above command causes the file `myoutput`, which is created if it does not exist, or rewound and truncated if it does, to be connected to the standard output of the program `myprog` when it is run. So if the FORTRAN 77 program `myprog` writes to unit 6, it writes to the file `myoutput`.

Output/Append

The shell command line for `myprog` to *append* to `mydata` is:

```
demo% myprog >> myoutput
```

The above command causes the file myoutput, which must exist, to be connected for appending. So if the FORTRAN 77 program myprog writes to unit 6, it writes to the file myoutput, but after wherever the file ended before.

You can redirect standard input and output on the same command line.

3.6 Piping

You can connect the standard output of one program directly to the standard input of another without using an intervening temporary file. The mechanism to accomplish this is called a *pipe*. Some consider piping to be a special kind of redirecting.

Example: A shell command line using a pipe:

```
demo% firstprog | secondprog
```

This command causes the standard output (unit 6) of firstprog to be piped to the standard input (unit 5) of secondprog. Piping and file redirection can be combined in the same command line.

Example: myprog reads mydata and pipes the output to wc; wc writes to datacnt.

```
demo% myprog < mydata | wc > datacnt
```

The program myprog takes its standard input from the file mydata, then pipes its standard output into the standard input of the wc command. The standard output of wc is then redirected into the file datacnt.

You can redirect standard error so it does not appear on your workstation display. In general, this is not a good idea, since you usually want to see error messages from the program immediately, rather than sending them to a file.

The shell syntax to redirect standard error varies, depending on whether you are using sh or csh.

Example: Redirecting and piping standard error and standard output in csh:

```
demo% myprog1 |& myprog2
```

Example: Redirecting and piping standard error and standard output in sh:

```
demo$ myprog1 2>&1 | myprog2
```

In each shell, the above command runs the program, myprog1, and redirects and pipes standard output and error to the program, myprog2.

Disk and Tape Files 4

This chapter is organized into the following sections.

4.1 File Access from FORTRAN 77 Programs

Data are transferred to or from devices or files by specifying a logical unit number in an I/O statement. Logical unit numbers can be nonnegative integers or the character `*`. `*` stands for the *standard input* if it appears in a `READ` statement, or the *standard output* if it appears in a `WRITE` or `PRINT` statement.

Standard input and standard output are explained in the section, "Preconnected Units" on page 121.

Accessing Named Files

Before a program can access a file with a `READ`, `WRITE`, or `PRINT` statement, the file must be created, and a connection established for communication between the program and the file. The file can already exist, or can be created at the time the program executes. The FORTRAN 77 `OPEN` statement establishes a connection between the program and the file to be accessed.

For a description of `OPEN`, read the chapter on statements in the *FORTRAN 77 4.0 Reference Manual*.

File names can be simple expressions, such as:

- Quoted character constants:

```
    FILE='myfile.out'
```

- Character variables:

```
    FILE=FILNAM
```

File names can be more complicated expressions, such as character expressions:

```
    FILE=PREFIX(:LNBLNK(PREFIX)) // '/' //
&        NAME(:LNBLNK (NAME)),…
```

A program can obtain file names in one of the following ways:

- By reading from a file or terminal keyboard, such as:

```
    READ( 4, 401) FILNAM
```

- From the command line, such as:

```
    CALL GETARG( ARGNUMBER, FILNAM )
```

- From the environment, such as:

```
    CALL GETENV( STRING, FILNAM )
```

This example shows one way to construct a file name in the C shell:

GetFilNam.f

This program uses the library routines getenv, lnblnk, and getcwd, which perform the functions of getting the environment, getting the last nonblank, and getting the current working directory, respectively.

```
      CHARACTER F*8, FN*40, FULLNAME*40
      READ *, F
      FN = FULLNAME( F )
      PRINT *, FN
      END

      CHARACTER*40 FUNCTION FULLNAME( NAME )
      CHARACTER NAME*(*), PREFIX*40
C         This assumes C shell.
C         Leave absolute path names unchanged.
C         If name starts with '~/', replace tilde with home
C         directory; otherwise prefix relative path name with
C         path to current directory.
      IF ( NAME(1:1) .EQ. '/' ) THEN
          FULLNAME = NAME
      ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
          CALL GETENV( 'HOME', PREFIX )
          FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&                  NAME(2:LNBLNK(NAME))
      ELSE
          CALL GETCWD( PREFIX )
          FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&                  '/' // NAME(:LNBLNK(NAME))
      ENDIF
      RETURN
      END
```

Compile and run GetFilNam.f as follows:

```
demo% f77 -silent GetFilNam.f
demo% a.out
"/hokey"
/hokey
demo%
```

Accessing Unnamed Files

When a program opens a FORTRAN 77 file without a name, the runtime system supplies a file name. There are several ways this is done.

Opened as Scratch

If you specify `STATUS='SCRATCH'` in the `OPEN` statement, then the system opens a file with a name of the form: `tmp.F`*AAAxnnnnn*, where *nnnnn* is replaced by the current process ID, *AAA* is a string of three characters, and *x* is a letter; the *AAA* and *x* make the file name unique. This file is deleted upon termination of the program or execution of a `CLOSE` statement, unless `STATUS='KEEP'` is specified in the `CLOSE` statement.

Already Open

If a FORTRAN 77 program has a file already open, an `OPEN` statement that specifies only the file’s logical unit number and the parameters to change can be used to change some of the file’s parameters; specifically, `BLANK` and `FORM`. The system determines that it must not really `OPEN` a new file, but just change the parameter values. Thus, this case looks like one where the runtime system would make up a name, but it does not.

Other

In all other cases, the runtime system `OPEN`s a file with a name of the form `fort.`*n*, where *n* is the logical unit number given in the `OPEN` statement.

Passing File Names to Programs

The file system does not have any notion of temporary file name binding or file equating, as some other systems do. File name binding is the facility that is often used to associate a FORTRAN 77 logical unit number with a physical file without changing the program. This mechanism evolved to communicate file names more easily to the running program, because in FORTRAN 66, you cannot open files by name.

With this operating system, there are several satisfactory ways to communicate file names to a FORTRAN 77 program.

- Command-line arguments and environment-variable values. For example, read the file `ioinit.f` in `libF77`. See the section, “Logical Unit Preattachment.” The program can then use those logical names to open the files.
- Redirection and piping. Chapter 3, “File System and FORTRAN 77 I/O,” describes *redirection* and *piping*, two other ways to change the program input and output files without changing the program.

Preconnected Units

When a FORTRAN 77 program begins execution under this operating system, there are usually three units already open. These are *preconnected units*. Their names are *standard input*, *standard output*, and *standard error*. In FORTRAN 77:

- Standard input is logical unit 5
- Standard output is logical unit 6
- Standard error is logical unit 0

All three are connected, unless file redirection or piping is done.

Other Units

All other units are preconnected to files named `fort.`*n*, where *n* is the corresponding unit number, and can be 0, 1, 2, …, with 0, 5, and 6 having the usual special meanings.

These files need not exist. They are created only if the units are actually used, and if the first action to the unit is a `WRITE` or `PRINT`; that is, only if an `OPEN` statement does not override the preconnected name before any `WRITE` or `PRINT` is issued for that unit.

Example: Preconnected files: the program `OtherUnit.f`:

```
    WRITE( 25, '(I4)' ) 2
    END
```

The above program preconnects the file `fort.25` and writes a single formatted record onto that file.

```
demo% f77 -silent OtherUnit.f
demo% a.out
demo% cat fort.25
   2
demo%
```

Logical Unit Preattachment

The `IOINIT` routine can also be used to attach logical units to specific files at runtime. It looks in the environment for names of a user-specified form, and then opens the corresponding logical unit for sequential formatted I/O. Names must be of the general form *PREFIXnn*, where the particular *PREFIX* is specified in the call to `IOINIT`, and *nn* is the logical unit to be opened. Unit numbers less than 10 must include the leading 0. See `IOINIT`(3F).

Example: Attach external files `test.inp` and `test.out` to units 1 and 2:

First, set the environment variables.

In `sh`:

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

In `csh`:

```
demo% setenv TST01 ini1.inp
demo% setenv TST02 ini1.out
```

The program `ini1.f` reads 1 and writes 2.

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

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

```
demo% cat ini1.inp
 12 3.14159012 6
demo% f77 -silent ini1.f
demo% a.out
demo% cat ini1.out
  12    3.14159  6
demo%
```

`IOINIT` is adequate for most programs as written. However, it is written in FORTRAN 77 specifically to serve as an example for similar user-supplied routines. Retrieve a copy as follows:

```
demo% cp /opt/SUNWspro/SC4.0/src/ioinit.f  .  (Solaris 2.x)
```

Logical File Names

If you are porting from VMS FORTRAN, the VMS style logical file names in the `INCLUDE` statement are mapped to UNIX path names. The environment variable `LOGICALNAMEMAPPING` defines the mapping between the logical names and the UNIX path name. If the environment variable `LOGICALNAMEMAPPING` exists, and if the `-xl[d]` compiler option is set, then the compiler interprets VMS logical file names on the `INCLUDE` statement.

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

```
"lname1=path1; lname2=path2; … "
```

Each *lname* is a logical name and each path is the path name of a directory (without a trailing /). All blanks are ignored when parsing this string. It strips any trailing `/[no]list` from the file name in the `INCLUDE` statement. Logical names in a file name are delimited by the first : in the VMS file name. The compiler converts file names of the form:

```
lname1:file
```

to:

```
path1/file
```

For logical names, uppercase and lowercase are significant. If a logical name is encountered on the `INCLUDE` statement, which is not specified in the `LOGICALNAMEMAPPING`, the file name is used, unchanged.

Direct I/O

Random access to files is also called direct access. A direct-access file contains a number of records that are written to or read from by referring to the record number. This record number is specified when the record is written. In a direct-access file, records must be all the same length and all the same type.

A logical record in a direct access, external file is a string of bytes of a length specified when the file is opened. `READ` and `WRITE` statements must not specify logical records longer than the definition of the original record size. Shorter logical records are allowed. Unformatted, direct writes leave the unfilled part of the record undefined. Formatted, direct writes cause the unfilled record to be padded with blanks.

When using direct unformatted I/O, be careful with the number of values your program expects to read. Each READ operation acts on exactly *one* record; the number of values that the input list requires must be *less than or equal to* the number of values in that record.

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

Example: Direct-access, *unformatted:*

```
    OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=20,
&       FORM='UNFORMATTED', ERR=90 )
    READ( 2, REC=13, ERR=30 ) X, Y
```

This program opens a file for direct-access, unformatted I/O, with a record length of 20 characters, then reads the thirteenth record as is.

Example: Direct-access, *formatted*:

```
    OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=20,
&       FORM='FORMATTED', ERR=90 )
    READ( 2, FMT="(I10,F10.3)", REC=13, ERR=30 ) A, B
```

This program opens a file for direct-access, formatted I/O, with a record length of 20 characters, then reads the thirteenth record and converts it according to the format:(I10,F10.3).

You can improve direct access I/O performance by opening a file with a large buffer size. Do this with one of the options for the OPEN statement, the FILEOPT=*fopt* option. ♦

fopt itself can be BUFFER=*n*. The form of the option is:

```
    OPEN( …, FILEOPT="BUFFER=n", … )
```

The option sets the size in bytes of the I/O buffer to use. For WRITEs, larger buffers yield faster I/O. For good performance, make the buffer a multiple of the largest record size. This size can be larger than actual physical memory; however, probably the very best performance is obtained by making the record size equal to the entire file size.

These larger buffer sizes may cause some extra paging. Read the section on the OPEN statement in the *FORTRAN 77 4.0 Reference Manual*.

Internal Files

An internal file is an object of type character such as a variable, substring, array, element of an array, or field of a structured record. If you are reading from the internal file, it can be a *constant* character string. This is called I/O, although I/O is not a precise term to use here, because you use READ and WRITE statements to deal with internal files.

- To use an internal file, give the name of the character object in place of the unit number.
- For a constant, variable, or substring, there is only a single record in the file.
- For an array, each array element is a record.
- f77 extends direct I/O to internal files. The ANSI standard includes only sequential formatted I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings.
- Each sequential READ or WRITE starts at the beginning of an internal file.

Example: Sequential formatted read from an internal file (one record only):

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

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

```
demo% cat intern3.f
    CHARACTER  LINE(4)*16
*                     12341234
    DATA  LINE(1) / ' 81  81 ' /
    DATA  LINE(2) / ' 82  82 ' /
    DATA  LINE(3) / ' 83  83 ' /
    DATA  LINE(4) / ' 84  84 ' /
    READ( LINE,'(2I4)') I,J,K,L,M,N !  This code reads an internal file.
    PRINT *, I, J, K, L, M, N
    END
demo% f77  -silent intern3.f
demo% a.out
   81  81  82  82  83  83
demo%
```

Example: Direct-access read from an internal file (one record):

```
demo% cat intern2.f
    CHARACTER LINE(4)*16
*                     12341234
    DATA  LINE(1) / ' 81  81 ' /
    DATA  LINE(2) / ' 82  82 ' /
    DATA  LINE(3) / ' 83  83 ' /
    DATA  LINE(4) / ' 84  84 ' /
    READ ( LINE, FMT=20, REC=3 ) M, N   !  This code reads an internal file.
20  FORMAT( I4, I4 )
    PRINT *, M, N
    END
demo% f77  -silent intern2.f
demo% a.out
   83  83
demo%
```

4.2 Tape I/O

Using tape files on UNIX systems is awkward because, historically, UNIX development was oriented toward small data sets residing on fast disks. Magnetic tape was used by early UNIX systems for archival storage and moving data between different machines. Unfortunately, many FORTRAN 77 programs are intended to use large data sets from magnetic tape.

For tape, it is more reliable to use the `TOPEN()` routines than the FORTRAN 77 I/O statements.

Using `TOPEN` for Tape I/O

A nonstandard tape I/O package (see `TOPEN` (3F)) offers a partial solution to the problem. You can transfer blocks between the tape drive and buffers declared as FORTRAN 77 character variables. You can then use internal I/O to fill and empty these buffers. This facility does not integrate with the rest of FORTRAN 77 I/O. It even has its own set of tape logical units.

For tapes, it is more reliable to use the `TOPEN()` routines than the FORTRAN 77 I/O statements.

FORTRAN 77 Formatted I/O for Tape

The FORTRAN 77 I/O statements provide facilities for transparent access to *formatted*, sequential files on magnetic tape. The tape block size can be optionally controlled by the `OPEN` statement `FILEOPT` parameter. There is no bound on formatted record size, and records may span tape blocks.

FORTRAN 77 Unformatted I/O for Tape

Using the FORTRAN 77 I/O statements to connect a magnetic tape for *unformatted* access is less satisfactory. Note the implementation of unformatted records as a sequence of characters preceded and followed by character counts. The size of a record (+ 8 characters of overhead) cannot be bigger than the buffer size.

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

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

Tape File Representation

A FORTRAN 77 data file is represented on tape by a sequence of data records followed by an `endfile` record. The data is grouped into blocks, the maximum size determined when the file is opened. The records are represented the same as records in disk files: formatted records are followed by newlines; unformatted records are preceded and followed by character counts. In general, there is no relation between FORTRAN 77 records and tape blocks; that is, records can span blocks, which can contain parts of several records.

The only exception is that FORTRAN 77 does not write an unformatted record that spans blocks; thus, the size of the largest unformatted record is eight characters less than the block size.

The `dd` *Conversion Utility*

An endfile record in FORTRAN 77 maps directly into a tape mark. In this respect, FORTRAN 77 files are the same as tape system files. But since the representation of FORTRAN 77 files on tape is the same as that used in the rest of UNIX, naive FORTRAN 77 programs cannot read 80-column card images from tape. If you have an existing FORTRAN 77 program and an existing data tape to read with it, translate the tape using the `dd`(1) utility, which adds newlines and strips trailing blanks.

Example: Convert a tape on `mt0` and pipe that to the executable `ftnprg`:

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

The `getc` Library Routine

If you write or modify a program, but do not want to use `dd`, you can use the `getc`(3F) library routine to read characters from the tape. You can then combine the characters into a character variable and use internal I/O to transfer formatted data. See also `TOPEN`(3F).

End-of-File

The end-of-file condition is reached when an endfile record is encountered during execution of a `READ` statement. The standard states that the file is positioned after the endfile record. In real life, this means that the tape read head is poised at the beginning of the next file on the tape. Thus, it would seem that you can read the next file on the tape; however, this is not true, and is not covered by the standard.

The standard also says that a `BACKSPACE` or `REWIND` statement can be used to reposition the file. Consequently, after reaching end-of-file, you can backspace over the endfile record and further manipulate the file, such as writing more records at the end, rewind the file, and reread or rewrite it.

Access on Multiple-File Tapes

Each tape drive can be opened by many names. The name used determines certain characteristics of the connection, which are the recording density and whether the tape is automatically rewound when opened and closed.

To access a file on a multiple-file tape, use the `mt`(1) utility to position the tape to the correct file, then open the file as a no-rewind magnetic tape, such as `/dev/nrmt0`. Using the tape with this name also prevents it from being repositioned when it is closed. If your program reads the file until end-of-file, then reopens it, it can access the next file on the tape. Any programs that follow can access the tape where you left it, preferably at the beginning of a file, or past the endfile record.

If your program terminates prematurely, it can leave the tape positioned anywhere.

Program Development 5

This chapter is organized into the following sections:

5.1 Simple Program Builds

For a program that depends on only a single source file and some system libraries, you compile all of the source files every time you change the program. Even in this simple case, however, executing the `f77` command can involve a lot of typing, and with options or libraries, a lot to remember. A script or alias can help.

Scripts or Aliases

You can write a shell script to save typing. For example, to compile a program in the file `example.f`, and which uses the SunCore® graphics library, you can save a one-line shell script onto a file called `fex`, that looks like this:

```
f77 example.f –lcore77 –lcore –o example
```

You may need to put execution permissions on `fex`:

```
demo% chmod +x fex
```

You can also create an alias for the same command:

```
demo% alias fex "f77 example.f –lcore77 –lcore –o example"
```

Either way, to recompile `example.f`, you type only `fex`:

```
demo% fex
```

Limitations

With multiple source files, forgetting one compile makes the objects inconsistent with the source. It is a time drain to recompile all the files after every editing session, since not every source file needs recompiling. Also, omitting an option or a library produces erroneous executables at times.

5.2 Program Builds with the `make` Program

The `make` program recompiles only what needs recompiling, and it uses only the options and libraries you want. This section shows you how to use normal, basic `make`, and provides a simple example. For a summary, see `make`(1).

The `makefile`

The way you tell `make` what files depend on other files, and what processes to apply to which files, is to put this information into a file, called the `makefile`, in the directory where you are developing the program.

For example, suppose you have a program of four source files and a `makefile`:

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

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

The `makefile` looks like this:

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

The first line of this `makefile` says: `make pattern`. `pattern` depends on `pattern.o`, `computepts.o`, and `startupcore.o`.

The second line is the command for making `pattern`.

The third line is a continuation of the second.

There are four such entries in this `makefile`. The structure of these entries is:

- **Dependencies**—Each entry starts with a line that names the file to `make`, and names all the files it depends on.
- **Commands**—Each entry has one or more subsequent lines that contain Bourne shell commands, which specify how to build the target file for this entry. These subsequent lines must each be indented by a tab.

make

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

```
demo% make
```

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

The `make` utility general actions are:

- From the `makefile`, it gets all the target files it must `make`, and what files they depend on. It also determines the commands used to `make` the target files.
- It gets the date and time each file was last changed.
- If any target file is not up-to-date with the files it depends on, then `make` rebuilds that target, using the commands from the `makefile` for that target.

The C Preprocessor

You can use the C preprocessor for such tasks as passing strings to `f77`.

For example, if you want your program to print the time it was compiled when it is given a command-line argument of `–v`, then you must add code that looks like this:

```
      IF (ARGSTRING .EQ. "-v") THEN
          PRINT *, CTIME
          CALL EXIT(0)
      END IF
```

This example is just an extension of the `make` example with `pattern.f`.

Use the C preprocessor to define `CTIME` as a quoted string that can be printed. The next two examples show how to do this.

The C preprocessor is applied if the file names have the suffix `.F`, so we change the file name:

```
demo% mv pattern.f pattern.F
```

The `–D` option defines a name to have a specified value for the C preprocessor, as if by a `#define` line. Consequently, we change the compilation line for `pattern.F` in the `makefile` as follows (in `sh` only):

```
demo% f77 "–DCTIME=\"‘date‘\"" –c –u pattern.F
```

The command line up to the `-c` option obtains the output of the `date` command, puts quotes around it, places that into `CTIME`, and passes it on to the C preprocessor. If you do not want the details, skip the next paragraph.

The innermost single quotes are backquotes or grave accents. They indicate that the output of the command contained in them (in this case, the date command) is to be substituted in place of the backquoted words. The next level of quote marks is what makes this define a FORTRAN 77 quoted string, so it can be used in the `print` statement. These marks must be escaped (or quoted) by preceding backslashes because they are nested inside another pair of quote marks. The outermost marks indicate to the interpreting shell that the enclosed characters are to be interpreted as a single argument to the `f77` command. They are necessary because the output of the `date` command contains blanks, so that, without the outermost quoting, it would be interpreted as several arguments, which would not be acceptable to `f77`.

The preprocessor now converts `CTIME` to `"jan15…"`, so that:

```
PRINT *, CTIME
```

becomes:

```
PRINT *, "jan15…"
```

The purpose here is to show how such strings are passed to the C preprocessor. The particular string passed is not useful, but the method is the same.

Macros with `make`

The `make` program does simple parameterless *macro* substitutions. In the `make` example above, the list of relocatable files that go into the target program `pattern` appears twice: once in the dependencies, and once in the `f77` command that follows. Doing so makes changing the `makefile` error-prone, since the same changes must be made in two places in the file.

Sample Macro Definition

You can add a macro definition to the beginning of your `makefile`, such as:

```
OBJ = pattern.o computepts.o startupcore.o
```

Sample Use of Macro Definition

Change the description of the program, `pattern` as follows:

```
pattern: $(OBJ)
    f77 $(OBJ) –lcore77 –lcore –lsunwindow \
    –lpixrect –o pattern
```

Note the special syntax in the above example: use of a macro is indicated by a dollar sign, immediately followed by the name of the macro in parentheses. For macros with single-letter names, you can omit the parentheses.

Overriding of Macro Values

The initial values of `make` macros can be overridden with command-line options to `make`. Add the following line to the top of the `makefile`:

```
FFLAGS=–u
```

Change each command for making FORTRAN 77 source files into relocatable files by deleting that flag,. The compile-line of `computepts.f` looks like this:

```
f77 $(FFLAGS) –c computepts.f
```

The final link looks like this:

```
f77 $(FFLAGS) $(OBJ) –lcore77 –lcore –lsunwindow \
   lpixrect –o pattern
```

If you issue the bare `make` command, everything compiles as before. However, the following command does more:

```
demo% make "FFLAGS=–u –O"
```

Here, the `-O` flag and the `-u` flag are passed to `f77`.

Suffix Rules in `make`

If you do not specify how to `make` a relocatable file, `make` uses one of its default rules. In this case, it uses the `f77` compiler, and passes as arguments any flags specified by the `FFLAGS` macro, the `-c` flag, and the name of the source file to be compiled.

You can take advantage of this rule twice in the example, but must still explicitly state the dependencies and the nonstandard command for compiling the `pattern.f` file. The `makefile` is as follows:

```
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=–u
pattern: $(OBJ)
    f77 $(OBJ) –lcore77 –lcore –lsunwindow \
    –lpixrect –o pattern
pattern.o: pattern.f commonblock
    f77 $(FFLAGS) "–DCTIME=\"‘date‘\"" –c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

5.3 Change Tracking and Control with SCCS

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

- Keep track of the evolution of a source file—its change history
- Prevent the same source file from being changed at the same time
- Keep track of the version number by providing version stamps

The basic three operations of SCCS are:

- Putting files under SCCS control
- Checking out a file for editing
- Checking in a file

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

Putting Files under SCCS

Putting files under SCCS control involves:

- Making the SCCS directory
- Inserting SCCS ID keywords into the files, an optional task
- Creating the SCCS files

Making the SCCS Directory

To begin, you must create the SCCS subdirectory in the directory in which your program is being developed. Use this command:

```
demo% mkdir SCCS
```

`SCCS` must be in uppercase.

Inserting SCCS ID Keywords

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

- Comment lines
- Parameter statements
- Initialized data

The advantage is that the version information appears in the compiled object program, and can be printed using the `what` command. Included header files that contain only parameter and data definition statements do not generate any initialized data, so the keywords for those files usually are put in comments or in parameter statements. Finally, in the case of some files, like ASCII data files or `makefile`s, the source is all there is, so the SCCS information can go in comments, if anywhere.

Identify the makefile with a `make` comment containing the keywords:

```
#   %Z%%M% %I% %E%
```

The source files, `startupcore.f`, `computepts.f`, and `pattern.f` can be identified by initialized data of the form:

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

You can also replace the word `CTIME` by a parameter that is automatically updated whenever the file is accessed with `get`.

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

Remove the `-DCTIME` option from the `makefile`. Finally, the included file `commonblock` is annotated with a FORTRAN 77 comment:

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

Creating SCCS Files

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

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

The makefile reads:

```
#   @(#)makefile1.184/03/01
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
    f77 $(OBJ) –lcore77 –lcore –lsunwindow \
        –lpixrect –o pattern
pattern.o: pattern.f commonblock
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

The `commonblock` file reads:

```
C   @(#)commonblock1.184/03/01
    INTEGER NMAX, NPOINTS
    REAL X, Y
    PARAMETER ( NMAX = 200 )
    COMMON NPOINTS
    COMMON X(NMAX), Y(NMAX)
```

The `computepts.f` file reads:

```
      SUBROUTINE COMPUTEPTS
      DOUBLE PRECISION T, DT, PI
      PARAMETER ( PI=3.1415927 )
      INCLUDE 'commonblock'
      INTEGER I
      CHARACTER*50 SCCSID
      DATA SCCSID/"@(#)computepts.f1.184/03/05\n"/
c     Compute x/y coordinates of NPOINTS points
c     on a unit circle as index I moves from 1 to
c     NPOINTS, parameter T sweeps from 0 to
c     PI(2 + NPOINTS/2) in increments of
c     (PI/2)*(1 + 4/NPOINTS)
      T = 0.0
      DT = (PI/2.0)*(1.0 + 4.0/DBLE(NPOINTS))
      DO 10 I = 1, NPOINTS+1
      X(I) = COS(T)
      Y(I) = SIN(T)
      T = T+DT
10    CONTINUE
      RETURN
      END
```

The `startupcore.f` file reads:

```
      SUBROUTINE STARTUPCORE
      INCLUDE '/usr/include/f77/usercore77.h'
C      Make initializing calls to core library
      COMMON /VWSURF/ VSURF
      INTEGER VSURF(VWSURFSIZE), SELECTVWSURF
      INTEGER PIXWINDD, INITIALIZECORE, INITIALIZEVWSURF
C      (Use CGPIXWINDD instead of PIXWINDD for color)
      EXTERNAL PIXWINDD
      CHARACTER*4 ENVRETURN
      CHARACTER*50 SCCSID
      INTEGER LOC
      DATA SCCSID/"@(#)startupcore.f 1.1 84/03/05\n"/
      DATA VSURF /VWSURFSIZE*0/

      VSURF(DDINDEX) = LOC(PIXWINDD)
      IF (INITIALIZECORE(BASIC, NOINPUT, TWOD) .NE. 0)
&     CALL EXIT
      CALL GETENV( "window_me", ENVRETURN )
      IF (ENVRETURN .EQ. " ") THEN
      WRITE(0,*) "Must run in a window"
      CALL EXIT(2)
      ENDIF
      IF (INITIALIZEVWSURF( VSURF, FALSE) .NE. 0)
&     CALL EXIT(2)
      IF (SELECTVWSURF(VSURF) .NE. 0) CALL EXIT(3)
      CALL SETWINDOW( -1.5, 1.5, -2.0, 2.0 )
      CALL CREATETEMPSEG()
      RETURN
      END

      SUBROUTINE CLOSECORE
      INCLUDE '/usr/include/f77/usercore77.h'
C     Make terminating calls to core library
      COMMON /VWSURF/ VSURF
      INTEGER VSURF(VWSURFSIZE)

      CALL CLOSETEMPSEG()
      CALL DESELECTVWSURF( VSURF )
      CALL TERMINATECORE()
      RETURN
      END
```

The `pattern.f` file reads:

```
      PROGRAM STAR
C     Draw a star of n points, arg n
      INCLUDE 'COMMONBLOCK'
      CHARACTER*10 ARG
      INTEGER I, IARGC, LNBLNK
      CHARACTER*(*) CTIME
      PARAMETER ( CTIME="84/03/05" )
      CHARACTER*50 SCCSID
      DATA SCCSID/"@(#)pattern.f1.184/03/05\n"/

      IF (IARGC() .LT. 1 ) THEN
          CALL GETARG( 0, ARG)
          I = LNBLNK(ARG)
          WRITE (0,*) "Usage: ",arg(:i)," -v or ",arg(:i)," nnn"
          CALL EXIT (0)
      END IF
      CALL GETARG( 1, ARG )
      IF (ARG .EQ. "-v") THEN
          PRINT *, CTIME
          CALL EXIT(0)
      END IF
      READ( ARG, '(I3)') NPOINTS
      NPOINTS = NPOINTS*4
      IF (NPOINTS .LE. 0 .OR. NPOINTS .GT. NMAX-1) THEN
          WRITE(0,*) NPOINTS/4, "Out of range [1..",(NMAX-1)/4,"]"
          CALL EXIT(12)
      END IF
      CALL COMPUTEPTS
      CALL STARTUPCORE
      CALL MOVEABS2( X(1),Y(1) )
      CALL POLYLINEABS2( X(2), Y(2), NPOINTS)
      PAUSE
      CALL CLOSECORE
      END
```

This is just an example of how SCCS operates, rather than how it is really used. You do not need the preprocessor any longer to drop in the compilation date. The `-v` argument is without purpose, since you can use the `what` command, which gives you much more detail.

Checking Files Out and In

Once your source code is under SCCS control, you use SCCS for two main tasks: to *check out* a file so that you can edit it, and to *check in* a file you have finished editing.

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

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

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

Check in the file with the `sccs delget` command when you have completed your editing. For example:

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

This command causes the SCCS system to do the following:

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

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

Libraries 6

This chapter is organized into the following sections:

6.1 Libraries in General

A software *library* is usually a set of subprograms. Each member of the set is called a library *element* or *module*. A *relocatable* library is one whose elements are relocatable (`.o`) files. These object modules are inserted into the executable file by the linker during the compile/link sequence. See `ld`(1).

There are two basic kinds of software libraries—static and dynamic:

- **Static library**—A library where modules are bound into the executable file *before* execution. Some examples on the system are:
  - FORTRAN 77 library: `libF77.a`
  - VMS FORTRAN 77 library: `libV77.a`
  - Math library: `libm.a`
  - C library: `libc.a`

- **Dynamic library**—A library where modules can be bound in *after* execution begins. Some examples on the system are:
  - FORTRAN 77 library: `libF77.so`
  - VMS FORTRAN 77 library: `libV77.so`
  - C library: `libc.so`

Advantages of Libraries

Relocatable libraries provide an easy way for commonly used subroutines to be used by several programs. You need only name the library when linking the program, and those library modules that resolve references in the program are linked—copied into the executable file.

There are two advantages:

- Only the needed modules are loaded.
- You need not change the link command line as subroutine calls are added and removed during program development.

Debug Aids

You can ask the linker various questions about libraries—how they are being used, what paths are being searched for libraries, and so forth.

Load Map

To display a load map, pass the load map option to the linker by `-Qoption`. This option displays which libraries are linked and which routines are obtained from which libraries during the creation of the executable module.

Example: `-m` for load map:

Solaris 2.x

```
demo% f77 -Qoption ld -m any.f77
```

Example: `-M` for load map:

Solaris 1.x

```
demo% f77 -Qoption ld -M any.f77
```

Other Queries

For Solaris 2.3 and later, there are linker debugging aids which help diagnose some linking problems. One way to get the list is `-Qoption ld -Dhelp`.

Example: List some linker debugging aid options:

Solaris 2.3

See the *Linker and Libraries Manual* in the Solaris documentation for details.

```
demo% f77 -Qoption ld -Dhelp any.f
     …
debug: files display input file processing (files and libraries)
debug: help  display this help message
debug: libs  display library search paths; detail flag shows
actual
debug:       library lookup (-l) processing
     …
demo%
```

Consistent Compile and Link

Do not build libraries with inconsistent options. Some options require consistent compiling and linking. Inconsistent compilation and linkage is not supported. See “Consistent Compile and Link,” on page 26, for the options and steps involved.

Fast Directory Cache for the Link-editor

Solaris 1.x

For Solaris 1.x only, the `ldconfig` utility configures a performance-enhancing cache for the `ld.so` runtime link-editor. It is run automatically from the `/etc/rc.local` file. For best performance, you should run it manually after you install a new shared object, such as a shared library, and every time the system is rebooted thereafter.

If you do not want to run `ldconfig` manually at each reboot of the system, add the name of the shared libraries directory to the `ldconfig` line near the end of the `rc.local` file. Do this on the machine where your compiler is installed, and on any client machines. Then run it manually once on each client.

Set the `ldconfig` path differently for standard and nonstandard installations:

- If you installed in the standard location, put that location in `rc.local`.
- If you installed into the nonstandard */your/dir/* location, use that path.

Example: Standard install—configure performance-enhancing cache:

```
demo% su root
Password: root-password
demo# vi /etc/rc.local
        ...
# Build the link-editor fast directory cache.
#
if [ -f /usr/etc/ldconfig ]; then
    ldconfig  /usr/lang/SC4.0; (echo "cache") > /dev/console
fi
:wq
demo#
```

In the above example, add the `/usr/lang/SC4.0/` directory to the `ldconfig` line near the end of the `rc.local` file. The `4.0` in `SC4.0` varies with the release number, of course.

Example: Nonstandard install—configure performance-enhancing cache:

```
...
ldconfig  /your/dir/SC4.0
...
```

6.2 Library Search Paths and Order

The linker searches for libraries in several locations in certain prescribed orders. Some of these locations are standard locations; some depend on the options `-lx` and `-Ldir`, and some on the environment variables `LD_RUN_PATH` or `LD_LIBRARY_PATH`. You can make some changes to the order and locations.

Order of Paths Critical for Compile (Solaris 1.x)

In Solaris 1.x, if you specify library search paths, the *order* of the paths can be critical. The compilation can fail if you cause an incompatible version of the math library, `libm`, to be used.

Symptom

If an entry is missing, the error message looks like the following:

```
ld: Undefined symbol
   _ _ _start_libm
   <other entries>
```

Solution

To fix the problem, use the correct order and get a compatible version of `libm`:

- If `/usr/lib` is in `LD_LIBRARY_PATH`, and if the installation was to `/usr/lang/` (*standard* installation), put `/usr/lang/lib` in `LD_LIBRARY_PATH` *before* `/usr/lib`
- If `/usr/lib` is in `LD_LIBRARY_PATH`, and if the installation was to /*my/dir/* (*nonstandard* installation), put /*my/dir/*`lib` in `LD_LIBRARY_PATH` before `/usr/lib`

Otherwise, an incompatible version of the math library, `libm`, is used.

Using `LD_LIBRARY_PATH` is not generally recommended.

Note – In Solaris 1.x, do not use `-L`*dir* to specify `/usr/lib`, because then you get an incompatible version of the math library, `libm`. You never need to use `-L`*dir* to specify `/usr/lib`, because you always get `/usr/lib` by default.

Error: `Library not Found`

In some circumstances, the dynamic linker cannot find some libraries.

Symptom

The runtime error message looks like this:

```
ld.so: library not found
```

This error happens while running of `a.out`, not during compilation or linking.

Some Causes

You may have created an executable using dynamic libraries, and moved the libraries. For example, you built `a.out` with your own dynamic libraries in /*my*/*libs*/, then moved the libraries.

You may also have replaced all the paths in `LD_LIBRARY_PATH` with one directory. For example, you defined the `LD_LIBRARY_PATH` environment variable to link in the XView libraries only.

Prevention

Set `LD_LIBRARY_PATH` to include the path where the missing library resides, instead of setting it to be only the one path.

Example: Put /*my*/*libs*/ into `LD_LIBRARY_PATH` in front of what is there:

In `sh`:

```
demo$ LD_LIBRARY_PATH=/my/libs/:$LD_LIBRARY_PATH
demo$ export LD_LIBRARY_PATH
```

In `csh`:

```
demo% setenv LD_LIBRARY_PATH /my/libs/:$LD_LIBRARY_PATH
```

Order on the Command Line for `–lx` *Options*

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

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

Search Order for Library Search Paths

Linker library search paths depend on the following:

- Solaris 1.x or 2.x
- Installation: standard location or nonstandard location, /*my*/*dir*/
- Building or running of the executable file

The base directory, here called *BaseDir*, is defined as follows:

	Standard Install	**Nonstandard Install to** /*my*/*dir*/
Solaris 1.x	`/usr/lang/`	/*my*/*dir*/
Solaris 2.x	`/opt/SUNWspro/`	/*my*/*dir*/`SUNWspro/`

While Building the Executable File

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

Solaris 1.x	/*BaseDir*/`lib/`	Sun shared libraries here
	/*BaseDir*/`SC4.0/lib/`	Sun libraries, shared or static, here
	`/usr/lang/lib/`	Standard location for Sun software
	`/usr/lib/`	Standard location for UNIX software

Solaris 2.x	/*BaseDir*/lib/	Sun shared libraries here
	/*BaseDir*/SC4.0/lib/	Sun libraries, shared or static, here
	/opt/SUNWspro/lib/	Standard location for Sun software
	/usr/ccs/lib/	Standard location for SVr4 software
	/usr/lib	Standard location for UNIX software

For both Solaris 1.x and 2.x, the above directories are the ones searched without any specification from you; they are the *default directories*.

While *building* the executable file in both Solaris 1.x and 2.x:

- The static linker searches paths specified by LD_LIBRARY_PATH. For the search order relative to the above paths, see ld(1).
- The static linker searches paths specified by -L*dir*. For the search order relative to LD_LIBRARY_PATH, see ld(1).

In general, it is best to avoid using LD_LIBRARY_PATH if at all possible.

While Running the Executable File

While running the executable file, the dynamic linker searches for *shared* libraries in these paths (among others), in the specified order

Solaris 1.x	/*BaseDir*/lib/	Sun shared libraries here
	/*BaseDir*/SC4.0/lib/	Sun libraries, shared or static, here
	/usr/lang/lib/	Standard location for Sun software
	/usr/lib/	Standard location for UNIX software
Solaris 2.x	/*BaseDir*/lib/	Built in by driver, unless -norunpath
	/opt/SUNWspro/lib	Built in by driver, unless -norunpath
	Other paths built in by -R or LD_RUN_PATH when the executable was generated	Uses paths stored in the executable. Ignores the current (runtime) values of -R and LD_RUN_PATH.
	/usr/lib/	Standard location for UNIX software

For both Solaris 1.x and 2.x, the above directories are the default directories, and are the ones searched without having to be specified.

***Remarks*—`LD_LIBRARY_PATH`*, *`LD_RUN_PATH`*, and* `-R`**

While *running* the executable file in either *Solaris 1.x and 2.x*:

- The *dynamic* linker searches paths specified by `LD_LIBRARY_PATH`. For the search order relative to the above paths, see `ld`(1).
- `LD_LIBRARY_PATH` can change after the executable has been created. No matter what the value of `LD_LIBRARY_PATH` was while the executable file was being built, the value at runtime is used while the executable is running. To see which directories were built in when the executable was created, use the `dump` command.

  Example: In *Solaris 2.x,* list the directories embedded in `a.out`:

```
demo% dump -Lv a.out | grep RPATH    (No comparable utility for 1.x)
```

While *running* the executable file in Solaris 2.x:

- The *dynamic* linker searches the paths that had been specified by `LD_RUN_PATH` or `-R` while the executable file was being generated.
- The current values of `LD_RUN_PATH` and `-R` are ignored. For `f77`, `-R` and `LD_RUN_PATH` are not identical; see *–R ls*, page 69, for the differences.

6.3 Static Libraries

Static libraries are built from object files (`.o` files) using the program, `ar`.

While the linker searches a static library, it extracts elements whose entry points are referenced in other parts of the program it is linking, such as subprogram or entry names or names of `COMMON` blocks initialized in `BLOCKDATA` subprograms. The nature of the elements and the nature of the search leads to some features that have both advantages and disadvantages.

Features of Libraries

There are three main features (advantages / disadvantages) of static libraries as compared to dynamic libraries:

- Static libraries are more self reliant and less adaptable.

  If you bind an `a.out` statically, then you can ship it without providing the libraries that were used to bind it. However, if there was a bug in a library that you bound into the `a.out`, then the statically bound `a.out` must be rebound and reshipped to take advantage of a fixed library. Whereas for dynamic libraries, the library provider can provide the fixed library to your customer, and not involve you.

- When the linker extracts a static library element, it takes the whole thing.

  Since an element corresponds to the result of a compilation, routines that are compiled together are always linked together. One result of this *whole-thing* approach is that if you compile a file that has many functions, then an `a.out` that uses only one of those functions gets all of them copied into and bound into the `a.out`.

  This is a difference between this operating system and some other systems, and may affect the way you divide up your libraries.

- In linking static libraries, the order really matters.

  The linker processes its input files in the order that they appear on the command line—left to right. When the linker decides whether or not a library element is to be linked, its decision is based only on the relocatable modules it has already processed.

  You can use `lorder` and `tsort` to order static libraries.

  Example: If the FORTRAN 77 program is in two files, `main.f` and `graf.f`, and only the latter accesses the SunCore graphics library, it is an error to reference that library before `graf.f` or `graf.o`:

```
demo% f77 main.f –lcore77 –lcore graf.f –o myprog (Incorrect)
demo% f77 main.f graf.f –lcore77 –lcore –o myprog (Correct)
```

Sample Creation of a Static Library

Example: Create a static library from four subroutines in one file:

Routines for library

```
demo% cat one.f
      subroutine twice ( a, r )
      real a, r
      r = a * 2.0
      return
      end
      subroutine half ( a, r )
      real a, r
      r = a / 2.0
      return
      end
      subroutine thrice ( a, r )
      real a, r
      r = a * 3.0
      return
      end
      subroutine third ( a, r )
      real a, r
      r = a / 3.0
      return
      end
demo%
```

Example: This main program uses one of the subroutines in the library:

Main

```
demo% cat teslib.f
      read(*,*) x
      call twice( x, z )
      write(*,*) z
      end
demo%
```

Split the file, using `fsplit`, so there is one subroutine per file:

```
demo% fsplit one.f
twice.f
half.f
thrice.f
third.f
demo%
```

Compile each with the `-c` option so it will compile only, and leave the `.o` files:

```
demo% f77 -c half.f
half.f:
half:
demo% f77 -c third.f
third.f:
third:
demo% f77 -c thrice.f
thrice.f:
thrice
demo% f77 -c twice.f
twice.f:
twice:
demo%
```

Create a static library, using `ar`:

```
demo% ar cr faclib.a half.o third.o thrice.o twice.o
```

The above command line directs `ar` to create static library `faclib.a` from the four object files.

As an alternative, specify any order using `lorder` and `tsort`:

```
demo% ar cr faclib.a 'lorder half.o third.o thrice.o \
             twice.o | tsort'
```

In Solaris 1.x, use `ranlib` to randomize the static library:

Solaris 1.x

```
demo% ranlib faclib.a                    ( Do not do this in Solaris 2.x)
```

To use this new library, put the file name in the compile command. No special flag is needed—the linker recognizes a library when it encounters one.

Example: Use the new library while compiling the main program:

```
demo% f77 teslib.f faclib.a       {Put the file name in the compile command. }
 teslib.f:
 MAIN:
 demo%
```

Example: Use `nm` to list the names of all the objects in the executable file:

This output format is for Solaris 2.x. It may vary for other releases.

twice appears→
half, third, thrice do not appear.

grep confirms that twice appears and half, third, thrice do not appear. →

```
demo% nm a.out
[Index]   Value      Size    Type  Bind  Other Shndx   Name
[1]  |         0|       0|FILE  |LOCL |0    |ABS    |a.out
…                        ← many lines not shown
[28] |    189024|       0|NOTY |LOCL |0    |13     |v.17
…                        ← many lines not shown
[190]|    193950|       1|OBJT |GLOB |0    |17     |__cblank
[191]|     77668|     164|FUNC |GLOB |0    |8      |MAIN_
…                        ← many lines not shown
[260]|     77832|      72|FUNC |GLOB |0    |8      |twice_
[261]|    194088|       4|OBJT |GLOB |0    |17
|_fp_current_exceptions
[262]|    188904|       0|FUNC |GLOB |0    |UNDEF  |close
[263]|    106400|      40|FUNC |GLOB |0    |8      |__rungetc
[264]|    113432|     784|FUNC |GLOB |0    |8      |__prnt_ext
[266]|    119624|     928|FUNC |WEAK |0    |8      |ieee_handler
…                        ← many lines not shown
demo% nm a.out | grep twice
[260]|     77832|      72|FUNC |GLOB |0    |8      |twice_
demo% nm a.out | grep half
demo% nm a.out | grep third
demo% nm a.out | grep thrice
demo%
```

Example: Test the executable file—run `a.out`:

```
demo% a.out
6
    12.0000
demo%
```

Sample Replacement in a Static Library

If you recompile an element of a static library, usually because you changed the source, replace it in its library by running `ar` again.

Example: Recompile, replace. Give `ar` the `r` option; use `cr` only for creating:

```
demo% f77 -c half.f
demo% ar r faclib.a half.o
demo%
```

6.4 Dynamic Libraries

The defining aspect of a *dynamic* library is that modules can be bound into the executable file *after* execution begins.

Perhaps the most useful feature of a dynamic library is that a module can be used by various executing programs *without* duplicating that module in each and every one of them. For this reason, a dynamic library is also called a *shared* library, or sometimes a *dynamic shared* library.

Features

A dynamic library has the following features:

- A dynamic library is a set of object modules, each in executable file format (the `a.out` format), but the set has no main entry.
- The object modules are *not* bound into the executable file by the linker during the compile-link sequence; such binding is deferred until runtime.

- A shared library module is bound once into the first running program that references it. If any subsequent running program references it, that reference is mapped to this first copy.
- If you change a module of a shared library, then whenever any application that uses it starts to execute, it uses the changed version. Maintaining programs is easier this way. However, a disadvantage is that you may have different results from an *unchanged* executable, or from what appears as an unchanged executable.

Performance Issues

There is the usual trade-off between space and time:

- **Less space**—In general, in deferring the binding of the library module:
  - A dynamic library uses less disk space.
  - A dynamic library uses less processor memory when several processes using the library are active simultaneously.
- **More time**—It takes a little more CPU time to do the following:
  - Load the library during runtime.
  - Do the link editing operations.
  - Execute the library position-independent code.
- **Possible time savings**—If the library module your program needs is already loaded and mapped because another running program referenced it, then the extra CPU time used can be offset by the savings in I/O access time. If the extra CPU time is less than or equal to the saved I/O time, then the performance that is the same or better.

  You can "get more bang for the buck" in an environment where multiple processes using the library are active simultaneously, that is, when the library is actually being shared. The extra bang comes from a reduction in working set size.

- **Overall speedup?** Programs vary. Because of these various performance issues, some programs are faster with shared libraries; some with nonshared libraries. You can bind each way to see if one way is significantly better for your program.

Position-Independent Code and `-pic`

Position-independent code (PIC) is code that can be bound to any address in a program without requiring relocation by the link editor. Since the code does not need the customizations created by such relocation, it is inherently sharable between multiple processors. Thus, if you are building code to be part of a shared library, you must make it position-independent code.

The `-pic` compiler option produces position-independent code. 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. The size of the global offset table is limited to 8K on SPARC processors. The `-PIC` compiler option is similar to `-pic`, but allows the global offset table to span the range of 32-bit addresses.

Binding Options

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

See “–Bx,” on page 41 and “–dx” on page 45.

If you provide a library to your customers, then providing both a dynamic and a static version allows them the flexibility of binding, whichever way is best for their application. For example, if the customer is doing some benchmarks, the `–dn` option reduces one element of variability.

A Simple Dynamic Library

If you compile the source files with `-pic` or `-PIC`, then you can build a dynamic library from the relocatable object (`.o`) files with the `ld` command.

Sample Create of a Dynamic Library (2.x)

Solaris 2.x

We can create a dynamic library, starting with the same files used for the static library example: `half.f`, `third.f`, `thrice.f`, and `twice.f`.

Example: Compile with `-pic`:

```
demo% f77 -pic -c -silent *.f
```

Example: Link and specify the `.so` file, and the `-h` to get a version number:

```
demo% ld -o libfac.so.1 -dy -G -h libfac.so.1 -z text *.o
```

`-G` tells the linker to build a dynamic library.

`-ztext` warns you if it finds anything other than position-independent code, such as relocatable text. It does not warn you if it finds writable data.

Example: Bind—make the executable file `a.out`:

```
demo% f77 teslib.f libfac.so.1
teslib.f:
 MAIN:
demo%
```

Example: Run:

```
demo% a.out
6
     12.0000
demo%
```

Inspect `a.out` for the use of shared libraries. The `file` command shows that `a.out` is a dynamically linked executable—programs that use shared libraries are completely link-edited during execution.

Example: Use the `file` command to see if `a.out` is dynamically linked:

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo% file a.out
a.out: ELF 32-bit MSB executable SPARC Version 1
dynamically linked, not stripped
demo%
```

The `ldd` command shows that `a.out` uses some shared libraries, including `libfac.so.1` and `libc`, which are included by default by `f77`. It also shows exactly which files on the system are used for these libraries.

Example: Use the `ldd` command to see if `a.out` uses shared libraries:

```
demo% ldd a.out
        libfac.so.1 =>   ./libfac.so.1
        libF77.so.2 =>   /opt/SUNWspro/lib/libF77.so.2
        libc.so.1 =>     /usr/lib/libc.so.1
        libucb.so.1 =>   /usr/ucblib/libucb.so.1
        libresolv.so.1 =>        /usr/lib/libresolv.so.1
        libsocket.so.1 =>        /usr/lib/libsocket.so.1
        libnsl.so.1 =>   /usr/lib/libnsl.so.1
        libelf.so.1 =>   /usr/lib/libelf.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libintl.so.1 =>  /usr/lib/libintl.so.1
        libw.so.1 =>     /usr/lib/libw.so.1
demo%
```

Your paths may vary.

Sample Create of a Dynamic Library (1.x)

Solaris 1.x

Start with the same files used for the static library example: `half.f`, `third.f`, `thrice.f`, `twice.f`. This library is very simple as it consists of procedures *only—no* global data is exported; it is made available for direct reference by programs using the library.

Example: Compile with `-pic`:

```
demo% f77 -silent -pic -c half.f third.f thrice.f twice.f
```

Example: Link, and specify the `.so` file and version number:

```
demo% ld -o libfac.so.1.1 -Bdynamic -assert pure-text *.o
```

`-assert pure-text` warns you if it finds anything other than position-independent code, such as relocatable text, but not if it finds writable data.

Example: Bind—make the executable file `a.out`:

```
demo% f77 teslib.f libfac.so.1.1
teslib.f:
 MAIN:
demo%
```

Example: Run:

```
demo% a.out
6
     12.0000
demo%
```

Inspect `a.out` for the use of shared libraries. The `file` command shows if `a.out` is a dynamically linked executable—programs that use shared libraries are completely link-edited while they are executed, that is, dynamically.

Example: Use the `file` command to see if `a.out` is dynamically linked:

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo% file a.out
a.out SPARC demand paged dynamically linked
      executable not stripped
demo%
```

The `ldd` command shows that `a.out` uses some shared libraries, including `libfac.so.1` and `libc` (included by default by `f77`). It also shows exactly which files on the system will be used for these libraries.

Example: Use the `ldd` command to see if `a.out` uses shared libraries:

```
demo% ldd a.out
        libfac.so.1.1
       -lF77.2 => /set/lang/4.0/lang/buildbin/4.x/libF77.so.2.0
        -lc.1 => /usr/lib/libc.so.1.6
demo%
```

Your paths may vary.

Dynamic Library for Exporting Initialized Data

Exported data means data in a shared library that is available for direct reference by programs using the library. For FORTRAN, exported initialized data is in the `COMMON` statements and the `BLOCK DATA` routines.

In Solaris 1.x, if the data are assigned initial values in the library, then this set of data must be identified for the link editor by placing the data (and *only* the data) in a special random archive library with the `.sa` suffix. No such step is needed in Solaris 2.x.

Solaris 1.x

To create a dynamic library that allows using initialized data, do the following:

1. **Segregate the initializing declarations into `BLOCK DATA` routines.**
2. **Put them in separate source files.**
3. **Create a static archive library (a `.sa` file) composed of only those routines.** You must include these modules in the `.so` file.
4. **Use `ranlib` to incorporate a symbol table into this `.sa` archive library.**

Note – The above steps are for Solaris 1.x only. In Solaris 2.x, it is all automatic.

Sample Create of a Dynamic Library—Export Initialized Data

Solaris 1.x

Example: Create dynamic library—allow exporting of initialized data:

```
demo% cat Blkgrp.f
* Blkgrp.f -- Block Data for Shared Library
      blockdata blkgrp
      common / grp / a, b, c
      data a, b, c / 3*9.9 /
      end
demo% cat PrintGrp.f
* PrintGrp.f -- Subroutine for Shared Library
      subroutine printgrp
      common / grp / a, b, c
      write( *, '(3f4.1)' ) a, b, c
      return
      end
demo% cat ReadGrp.f
* ReadGrp.f -- Subroutine for Shared Library
      subroutine readgrp
      common / grp / a, b, c
      read( *, * ) a, b, c
      return
      end
demo% cat TesSharMain.f
* TesSharMain.f -- Test Shared Library
      common / grp / a, b, c
      a = 1.0
      b = 2.0
      call printgrp
      end
demo%
```

Example: Source that *does* export initialized data:

```
demo% cat ZapGrp.f
* ZapGrp.f -- Subroutine for Shared Library
      subroutine zapgrp
      common / grp / a, b, c
      a = 0.0
      b = 0.0
      c = 0.0
      return
      end
demo%
```

Example: Use `-pic` on: `blkgrp.f`, `printgrp.f`, `readgrp.f`, and `zapgrp.f`:

```
demo% f77 -c -pic -silent *.f
demo%
```

Example: Create the `.sa` file, then run `ranlib` on it:

```
demo% ar cr libblkgrp.sa.1.1 Blkgrp.o
demo% ranlib libblkgrp.sa.1.1
demo%
```

Create a shared library `.so` file with the same version number as the `.sa` file. For the dynamic loader, the `.sa` and `.so` files must match exactly in name and version number.

Example: Create a shared library:

```
demo% ld -o libblkgrp.so.1.1 -assert pure-text \
      PrintGrp.o ReadGrp.o ZapGrp.o Blkgrp.o
demo%
```

Example: Bind:

```
demo% f77 TesSharMain.o libblkgrp.so.1.1
demo%
```

Example: Run:

```
demo% a.out
 1.0 2.0 9.9
demo%
```

Inspect the a.out file for the use of shared libraries. The file command shows that a.out is a dynamically linked executable—programs that use shared libraries are completely link-edited while they are executed, that is, dynamically.

Example: Use the file command to see if a.out is dynamically linked:

The output varies slightly for Solaris 1.x, 2.x, x86.

```
demo% file a.out
a.out: SPARC demand paged dynamically linked
     executable not stripped
demo%
```

The ldd command shows that a.out uses two shared libraries, libfac.so.1.1 and libc, which are included by default by f77. It also shows exactly which files on the system are used for these libraries.

Example: Use the ldd command to see if a.out uses shared libraries:

```
demo% ldd a.out
     libblkgrp.so.1.1
     -lF77.2 => /set/lang/2.0/lang/buildbin/4.x/libF77.so.2.0
     -lc.0 => /usr/lib/libc.so.0.10
demo%
```

6.5 Libraries Provided with the Compiler

Several libraries are installed with the compiler, including the following:

Table 6-1 Major Libraries Provided with the Compiler

Library	**File**	**Options Needed**
`f77` functions, nonmath	`libF77`	None
`f77` functions, nonmath, multithread safe	`libF77_mt`	`-parallel`, *and so on*
`f77` math library	`libM77`	None
VMS library	`libV77`	`-lV77`
Library used if linking Pascal, FORTRAN, and C objects	`libpfc`	None
Library of Sun math functions	`libsunmath`	None
POSIX bindings	`libFposix`	`-lFposix`
POSIX bindings for extra runtime checking	`libFposix_c`	`-lFposix_c`

XView bindings and Xlib bindings for the X11 interface	`libFxview`	`-lFxview`
		`-lxview`
		`-lX11`

VMS Library

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

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

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

See the *FORTRAN 77 4.0 Reference Manual* for details on these routines.

POSIX *Library*

There are two versions of POSIX bindings provided with the compiler:

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

If you pass bad handles:

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

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

Both POSIX libraries come in static and dynamic forms.

Which POSIX

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

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

POSIX.1 documents:

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

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

For further information, copies of the IEEE and ISO POSIX.1 Standard (ISO 9945-1:1990, also known as IEEE Standard 1003.1-1990) can be obtained from the following organizations:

- **Continental U.S.**:
  Computer Society: +1 (714) 821 8380 (Ask for Customer Service)
  or IEEE Publication Sales +1 (800) 678-IEEE

- **Canada**:
  IEEE Canada: +1 (908) 981-1393
  7071 Yonge St.
  Thornhill, Ontario L3T 2A6
  Canada

- **Outside Continental U.S.**:
  IEEE Service Center: +1 (800) 678-IEEE
  445 Hoes Lane
  P. O. Box 1331
  Piscataway, NJ 08855-1331

  or:

  IEEE Computer Society: +1 (714) 821 8380; Fax: +1 (714) 821 4010
  10662 Los Vaqueros Circle
  P. O. Box 3014
  Los Alamitos, CA 90720-3014

- **Europe**:
  IEEE Computer Society: +32 2 770 2198; Fax +32 2 770 8505
  Jacques Kevers
  13 Ave de l'Aquilon
  B-1200
  Brussels
  Belgium

- **Asia**:
  IEEE Computer Society: +81 33 408 3118; Fax +81 33 408 3553
  Ms. Kyoko Mikami
  Ooshima Building
  2-19-1 Minami Aoyma
  Minato-Ku
  Tokyo 107
  Japan

6.6 Shippable Libraries

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

Standard install	`/opt/SUNWspro/READMEs/runtime.libraries`
Install to */my/dir/*	*/my/dir/*`SUNWspro/READMEs/runtime.libraries`

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

Refer to the section, "*License to Use,*" in the document, "*End User Object Code License*," at the back of the plastic case that contains the CD-ROM.

Debugging 7

This chapter is organized into the following sections:

7.1 Global Program Checking (`-Xlist`)

Checking across routines helps find various kinds of bugs.

With `-Xlist`, `f77` reports errors of alignment, agreement in number and type for arguments, common blocks, parameters, plus many other kinds of errors.

`f77` also makes a listing and a cross reference table; combinations and variations of these are available using suboptions. An example follows.

Example: Use `-XlistE` to show *errors only*:

-XlistE

```
demo% f77 -XlistE -silent Repeat.f
demo% cat Repeat.lst
FILE  "Repeat.f"
program  repeat
     4            CALL nwfrk ( pn1 )
                                  ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is
                 integer*4
                 See: "Repeat.f" line #14
     4            CALL nwfrk ( pn1 )
                                  ^
**** ERR  #317:  variable "pn1" referenced as integer*4 across
                 repeat/nwfrk//prnok in line #21 but set as real
                 by repeat in line #2
subroutine  subr1
    10             CALL subr1 ( x * 0.5 )
                             ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #3
subroutine  nwfrk
    17            PRINT *, prnok ( ix ), fork ( )
                                     ^
**** ERR  #418:  argument "ix" is integer*4, but dummy argument
                 is real
                 See: "Repeat.f" line #20
subroutine  unreach_sub
    24           SUBROUTINE unreach_sub()
                                      ^
**** WAR  #338:  subroutine "unreach_sub" isn't called from program

Date:     Wed Feb 23 10:40:32 1995
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo%
```

Errors in General

Global program checking performs the following tasks:

- Enforce type checking rules of FORTRAN 77 more stringently than usual, especially between separately compiled routines.
- Enforce some portability restrictions needed to move programs between different machines or operating systems
- Detect legal constructions that are nevertheless wasteful or error-prone
- Reveal other bugs and obscurities

Details

In particular, global cross checking reports problems, such as:

- Interface problems
  - Checking number and type of dummy and actual arguments
  - Checking type of function values
  - Checking possible conflicts of incorrect usage of data types in common blocks of different subprograms
- Usage problems
  - Function used as a subroutine or subroutine used as a function
  - Declared but unused functions, subroutines, variables, and labels
  - Referenced but not declared functions, subroutines, variables, and labels
  - Usage of unset variables
  - Unreachable statements
  - Implicit type variables
  - Inconsistency of the named common block lengths, names, and layouts
- Syntax problems: syntax errors found in a FORTRAN 77 program
- Portability problems: code that does not conform to ANSI FORTRAN 77, if the appropriate option is used

How to Use Global Program Checking

To cross-check the named source files, use `-Xlist` on the command line.

Example: Compile three files for global program checking:

```
demo% f77 -Xlist  any1.f  any2.f  any3.f
```

In the above example, `f77`:

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

Terminal Output

To display directly to the terminal, rename the output file to `/dev/tty`.

Example: Display to terminal:

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

See `-Xlisto` *name*, on page 185.

Default Output Features

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

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

File Types

The checking process recognizes all the files in the f77 command line, which contain names that end in .f, .for, .F, .o, or .s. The .o and .s files supply the process with information that relates to global names only, such as subroutine and function names.

Analysis Files (.fln Files)

f77 stores results of local cross checking analysis for source files into files with a .fln suffix. It usually uses the source directory. The files may be clutter, however. One workaround is to delete the files from time to time:

```
demo% rm *.fln
```

Alternatively, put the files into, say, /tmp. See -Xlistfln*dir*, page 184.

```
demo% f77 -Xlistfln/tmp *.f
```

Example: Using `-Xlist`—a program with inconsistencies between routines:

`Repeat.f`

```
demo% cat Repeat.f
      PROGRAM repeat
        pn1 = REAL( LOC ( rp1 ) )
        CALL subr1 ( pn1 )
        CALL nwfrk ( pn1 )
        PRINT *, pn1
      END ! PROGRAM repeat

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

      SUBROUTINE nwfrk( ix )
        EXTERNAL fork
        INTEGER prnok, fork
        PRINT *, prnok ( ix ), fork ( )
      END

      INTEGER FUNCTION prnok ( x )
        prnok = INT ( x ) + LOC(x)
      END

      SUBROUTINE unreach_sub()
        CALL sleep(1)
      END
demo% f77 -Xlist -silent Repeat.f
demo% cat Repeat.lst
```

Compile with `-Xlist`. →
List the `-Xlist` output file. →

See the output on the following pages.

Example: Output file for `-Xlist`:

`Repeat.lst`

Error messages are embedded in the source listing.

```
FILE  "Repeat.f"
     1          PROGRAM repeat
     2            pn1 = REAL( LOC ( rp1 ) )
     3            CALL subr1 ( pn1 )
     4            CALL nwfrk ( pn1 )
                               ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer*4
                 See: "Repeat.f" line #14
     4            CALL nwfrk ( pn1 )
                               ^
**** ERR  #317:  variable "pn1" referenced as integer*4 across
             repeat/nwfrk//prnok in line #21 but set as real by repeat in
                 line #2
     5            PRINT *, pn1
     6          END ! PROGRAM repeat
     7
     8          SUBROUTINE subr1 ( x )
     9            IF ( x .GT. 1.0 ) THEN
    10             CALL subr1 ( x * 0.5 )
                          ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #3
    11            END IF
    12          END
    13
    14          SUBROUTINE nwfrk( ix )
    15            EXTERNAL fork
    16            INTEGER prnok, fork
    17            PRINT *, prnok ( ix ), fork ( )
                                  ^
**** ERR  #418:  argument "ix" is integer*4, but dummy argument is real
                 See: "Repeat.f" line #20
    18          END
    19
    20          INTEGER FUNCTION prnok ( x )
    21            prnok = INT ( x ) + LOC(x)
    22          END
    23
    24          SUBROUTINE unreach_sub()
                                   ^
**** WAR  #338:  subroutine "unreach_sub" isn't called from program
    25            CALL sleep(1)
    26          END
```

Output File: `f77 -Xlist Repeat.f` (*Continued*)

`Repeat.lst` ***(Continued)***

Cross reference table

Sample Interpretation:

The routine `nwfrk` →
called in `repeat`, line 4
defined, line 14

```
         C R O S S   R E F E R E N C E   T A B L E
  Source file:   Repeat.f
Legend:
D        Definition/Declaration
U        Simple use
M        Modified occurrence
A        Actual argument
C        Subroutine/Function call
I        Initialization: DATA or extended declaration
E        Occurrence in EQUIVALENCE
N        Occurrence in NAMELIST

         P R O G R A M   F O R M
 Program
 -------
repeat          <repeat>        D      1:D

 Functions and Subroutines
 -------------------------
fork     int*4  <nwfrk>        DC     15:D     16:D     17:C

int      intrinsic
                <prnok>         C     21:C

loc      intrinsic
                <repeat>        C      2:C
                <prnok>         C     21:C

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

prnok    int*4  <nwfrk>        DC     16:D     17:C
                <prnok>        DM     20:D     21:M

real     intrinsic
                <repeat>        C      2:C

sleep           <unreach_sub>           C     25:C

subr1           <repeat>        C      3:C
                <subr1>        DC      8:D     10:C
unreach_sub     <unreach_sub>           D     24:D
```

Output File: `f77 -Xlist Repeat.f` (*Continued*)

`Repeat.lst`
(Continued)

More of the cross reference table

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

ix       int*4  dummy
                <nwfrk>        DA     14:D     17:A

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

rp1      real*4 <repeat>        A      2:A

x        real*4 dummy
                <subr1>        DU      8:D      9:U     10:U
                <prnok>       DUA     20:D     21:A     21:U

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

Date:     Tue Feb 22 13:15:39 1995
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo%
```

In the cross-reference table in the above example:

- `ix` is a 4-byte integer:
  - Used as an argument in the routine, `nwfrk`
  - At line 14, used as a declaration of argument
  - At line 17, used as an actual argument
- `pn1` is a 4-byte real in the routine, `repeat:`
  - At line 2, modified
  - At line 3, argument
  - At line 4, argument
  - At line 5, used
- `rp1` is a 4-byte real in the routine, `repeat`. At line 2, it is an argument.
- `x` is a 4-byte real in the routines, `subr1` and `prnok`:
  - In `subr1`, at line 8, defined; at lines 9 and 10 used
  - In `prnok`, at line 20, defined; at line 21, used as an argument

Suboptions for Global Checking Across Routines

The standard global cross checking option is `-Xlist` with no suboption.

This section shows the listing, errors and cross reference table. For variations from this standard report, add one or more suboptions to the command line.

Suboption Syntax

Add suboptions according to the following rules:

- Append the suboption to `-Xlist`.
- Put no space between the `-Xlist` and the suboption.
- Put only one suboption per `-Xlist`.

Combination Special and A La Carte Suboptions

Combine suboptions according to the following rules:

- The combination special is: `-Xlist` (listing, errors, cross reference table)
- The a la carte options are: `-Xlistc`, `-XlistE`, `-XlistL`, and `-XlistX`.
- All other options are detail options—not a la carte or combination special.

Note – Once you start ordering a la carte, the three parts of the combination special are cancelled, and you get only what you specify.

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

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

```
 demo% f77  -Xlistc  any.f
```

The following table shows the combination special or a la carte suboptions, with no other suboptions:

Type/Amount of Output	Option	Comment	Details
Errors, listing, cross reference table	`-Xlist`	No suboptions	*page 176*
Errors	`-XlistE`	By itself, does not show listing or cross reference table	*page 184*
Errors and *listing*	`-XlistL`	By itself, does not show cross reference table	*page 185*
Errors and *cross reference table*	`-XlistX`	By itself, does not show listing	*page 186*
Errors and *call graph*	`-Xlistc`	By itself, does not show listing or cross reference table	*page 184*

Here is a summary of `-Xlist` suboptions:

Option	Action	Details
`-Xlist` (*no suboption)*	Show errors, listing, and cross reference table.	*page 182*
`-Xlistc`	Show call graphs and errors.	*page 184*
`-XlistE`	Show errors.	*page 184*
`-Xlisterr[`*nnn*`]`	Suppress error *nnn* in the verification report.	*page 184*
`-Xlistf`	Produce fast output.	*page 184*
`-Xlistfln`*dir*	Put the `.fln` files in *dir*.	*page 184*
`-Xlisth`	Halt the compilation if errors occur in cross-checking.	*page 185*
`-XlistI`	List and cross-check `include` files.	*page 184*
`-XlistL`	Show the listing and errors.	*page 185*
`-Xlistl`*n*	Set page breaks.	*page 185*
`-Xlisto` *name*	Rename the `-Xlist` output report file.	*page 185*
`-Xlists`	Suppress unreferenced identifiers from the cross reference table.	*page 185*
`-Xlistv`*n*	Show different amounts of semantic information.	*page 186*
`-Xlistw[`*nnn*`]`	Set the width of output lines.	*page 186*
`-Xlistwar[`*nnn]*	Suppress warning *nnn* in the report.	*page 186*
`-XlistX`	Show the cross-reference table and errors.	*page 186*

Details of `-Xlist` *Suboptions*

-Xlistc Show call graphs (and cross-routine errors). This suboption by itself does not show a listing or cross-reference. It produces the call graph in a planned tree form, using printable characters. If some subroutines are not called from `MAIN`, more than one graph is shown. Each `BLOCKDATA` is printed separately with no connection to `MAIN`.

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

-XlistE Show cross-routine errors. This suboption by itself does not show a listing or a cross reference.

-Xlisterr[*nnn*] Suppress error *nnn* in the verification report. This option is useful if you want a listing or cross-reference without the error messages. It is also useful if you do not consider certain practices to be real errors.

To suppress more than one error, use this option repeatedly. For example: `-Xlisterr338` suppresses error message 338. If *nnn* is not specified, all error messages are suppressed.

-Xlistf For faster output, produce source file listings and cross-checking and verify sources, but do not generate object files.

The default is: generate object files.

-Xlistfln*dir* Put the `.fln` files into the *dir* directory, which must already exist.

The default is the source directory.

-XlistI Include files. List and cross-check the include files.

If `-XlistI` is the only `-Xlist` option or suboption used, then you get the standard `-Xlist` output of a line numbered listing, error messages, and a cross-reference table, but `include` files are shown or scanned, as appropriate.

- **Listing**—If the listing is not suppressed, then the `include` files are listed in place. Files are listed as often as they are included. The files are:
  - Source files
  - `#include` files
  - `INCLUDE` files

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

  The default is no `include` files.

`-Xlist`**h** — Halt the compilation if errors are detected while cross-checking the program. In this case, the report is redirected to `stdout` instead of the `*.lst` file.

`-Xlist`**L** — Show listing and cross-routine errors. This suboption by itself does not show a cross reference. The default is to show the listing and cross-reference.

`-Xlist`**l***n* — Set the page length for pagination to *n* lines. The suboption is the letter *ell* for length, not the digit *one*. For example, `-Xlistl45` sets the page length to 45 lines. The default is 66.

The `-Xlistl0` option shows listings and cross-reference with no page breaks for easier on-screen viewing. The suboption is a *zero*, not a letter *oh*.

`-Xlist`**o** *name* — Rename the `-Xlist` output report file. The space between `o` and *name* is required. Output is then to the *name*`.lst` file.

To display directly to the terminal, use the command: `-Xlisto /dev/tty`

`-Xlist`**s** — Suppress unreferenced identifiers from the cross-reference table.

If the identifiers are defined in the `include` files but not referenced in the source files, then they are not shown in the cross-reference table.

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

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

-Xlistv*n* Set level of checking strictness. *n* is 1, 2, 3, or 4. The default is 2 (-Xlistv2).

- -Xlistv1

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

- -Xlistv2

  Show cross-checked information with summaries and line numbers. This is the normal level of checking strictness, and includes argument inconsistency errors and variable usage errors.

- -Xlistv3

  Show cross-checking with summaries and line numbers. Additionally to -Xlistv2, show common block maps. This is a high level of checking strictness, and includes errors caused by incorrect usage of data types in common blocks in different subprograms.

- -Xlistv4

  Show cross-checking with summaries and line numbers. Additionally to -Xlistv2, show common block maps and equivalence block maps. This is the top level of checking strictness with maximum error detection.

-Xlistw[*nnn*] Set width of output line to *n* columns. For example, -Xlistw132 sets the page width to 132 columns. The default is 79.

-Xlistwar[*nnn*] Suppress warning *nnn* in the report. If *nnn* is not specified, then all warning messages are suppressed from printing. To suppress more than one, but not all warnings, use this option repeatedly. For example, -Xlistwar338 suppresses the warning message, number 338.

-XlistX Show cross-reference table and cross-routine errors. This suboption by itself does not show a listing.

The cross-reference table shows the following information about each identifier:

- Is it an argument?
- Does it appear in a COMMON or EQUIVALENCE declaration?
- Is it set or used?

Example: Use `-Xlistwar`*nnn* to suppress two specific warnings:

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

Date:     Wed Feb 23 10:40:32 1995
Files:         2 (Sources: 1; libraries: 1)
Lines:        26 (Sources: 26; Library subprograms:2)
Routines:      5 (MAIN: 1; Subroutines: 3; Functions: 1)
Messages:      5 (Errors: 3; Warnings: 2)
demo%
```

Some warnings that are popular to suppress are: 314, 315, 320, 357.

Example: Explain a message and find a type mismatch:

`ShoGetc.f`

Type Z on keyboard →

The problem:
Why this message? →

The debugging:
Use `-Xlist`.
List the output.

Here is the error. →
Our default typing of `getc` is not consistent with the FORTRAN 77 library.

`f77` was given special information about the FORTRAN 77 library—that is how `f77` knows that `getc` is integer.

The solution: →
Make c an integer.

No more message.

```
demo% cat ShoGetc.f
     CHARACTER*1 c
     i = getc(c)
     END
demo% f77 -silent ShoGetc.f
demo% a.out          Program waits for input from keyboard
Z
 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Inexact;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
demo% f77 -XlistE -silent ShoGetc.f
demo% cat ShoGetc.lst
FILE  "ShoGetc.f"
program  MAIN
     2          i = getc(c)
                ^
**** WAR  #320:  variable "i" set but never referenced
     2          i = getc(c)
                       ^
**** ERR  #412:  function "getc" used as real but declared as
                 integer*4
     2          i = getc(c)
                         ^
**** WAR  #320:  variable "c" set but never referenced

Date:     Fri Mar  4 12:13:11 1995
Files:         2 (Sources: 1; libraries: 1)
Lines:         3 (Sources: 3; Library subprograms:1)
Routines:      1 (MAIN: 1)
Messages:      3 (Errors: 1; Warnings: 2)
demo% cat ShoGetc.f
     INTEGER c
     i = getc(c)
     END
demo% f77 -silent ShoGetc.f
demo% a.out
Z
demo%
```

7.2 Special Compiler Options (-C, -u, -U, -V, -xld)

The compiler options -C, -u, -U -V, and -xld are useful for debugging. They check subscripts, spot undeclared variables, show stages of the compile-link sequence, versions of software, and compile D debug statements.

For Solaris 2.3 and later, there are new linker debugging aids. See ld(1), or type: -Qoption ld -Dhelp.

Subscript Bounds (-C)

To check for out-of-bounds array subscripts, use -C.

If you compile with -C, then f77 checks at runtime for out-of-bounds on each array subscript. This action helps catch some causes of the segmentation fault.

Example: Index out of range:

```
demo% cat indrange.f
    REAL a(10,10)
    k = 11
    a(k,2) = 1.0
    END
demo% f77 -C -silent indrange.f
demo% a.out
 Subscript out of range on file indrange.f, line 3, procedure
MAIN.
 Subscript number 1 has value 11 in array a.
 Abort (core dumped)
demo%
```

Undeclared Variable Types (-u)

To check for any undeclared variables, use -u.

The -u option causes all variables to be initially identified as undeclared, so that an error is flagged for variables that are not explicitly declared. The -u flag is useful for discovering mistyped variables. If -u is set, all variables are treated as undeclared until explicitly declared. Use of an undeclared variable is accompanied by an error message.

Case-Sensitive Variable Recognition (-U)

To distinguish between uppercase and lowercase, use -U.

If you debug FORTRAN 77 programs that use other languages, you may need to compile with the -U option to preserve the case.

With the -U option, f77 does *not* convert uppercase letters to lowercase, but leaves them in the original case. The default is to convert to lowercase, except within character-string constants.

You need this option if the routine in the other language names a function or a common block with one or more uppercase letters. However, since -U also makes variable recognition case-sensitive, you must have perfect consistency as you use uppercase or lowercase for variable names and global identifiers.

Note – If you are not perfectly consistent with the case of your variables, the -U option will probably cause serious problems. That is, if you sometimes type Delta, and other times, DELTA or delta, then with -U, f77 treats these various *deltas* as totally different variables. This is probably not what you intend, and can waste debugging time.

Version Checking (-V)

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

D Comment Line Debug Print Statements (-xld)

To compile with comment line debug print statements, use -xld.

The -xld flag causes f77 to compile statements (usually print statements) that have a D or a d in column one. Without -xld, they are comments. See Section 2.9, "Directives," for details on -xl[d].

Note – The `-xld` option enables VMS FORTRAN compatibility mode, which may *not* be what you want, however. It is safe to use only if you normally compile with `-xl`, since it changes FORTRAN 77 semantics.

Example: Compile with and without `-xld`:

```
      REAL A(5) / 5.0, 6.0, 7.0, 8.0, 9.0 /
      DO I = 1, 5
           X = A(I)**2
D          PRINT *, I, X {With -xld, this prints I and X. Without, it prints nothing.}
      END DO
      PRINT *, 'done'
      END
```

7.3 The Debugger

This section introduces some `dbx` features likely to be used with `f77`. Use it as a quick start for `f77` debugging. This section is organized as follows:

Note – Before you use the debugger, you must install the appropriate Tools package—read *Installing SunSoft Developer Products (SPARC/Solaris)* for details.

Sample Program for Debugging

Here is a program that includes the files, `a1.f`, `a2.f`, and `a3.f`, that contain bugs, and is used in several examples of debugging.

Example: Main for debugging:

a1.f

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

Example: Subroutine for debugging:

a2.f

```
      SUBROUTINE mkidentity ( array, m )
      REAL array(m,m)
      DO 90 i = 1, m
           DO 20 j = 1, m
               IF ( i .EQ. j ) THEN
                array(i,j) = 1.
               ELSE
                array(i,j) = 0.
               END IF
 20        CONTINUE
 90   CONTINUE
      RETURN
      END
```

Example: Function for debugging:

a3.f

```
      REAL FUNCTION determinant ( a )
      REAL a(2,2)
      determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
      RETURN
      END
```

Sample dbx *Session*

The following examples use the sample program.

- Compile and link with the `-g` flag. You can do this in one or two steps.

  Example: Compile and link *in one step,* with `-g`:

```
demo% f77 -o silly -g a1.f a2.f a3.f
```

  Example: Compile and link *in separate steps*:

```
demo% f77 -c -g a1.f a2.f a3.f
demo% f77 -g -o silly a1.o a2.o a3.o (Use -g  in Solaris 1.x, but not in 2.x)
```

- To start `dbx`, type `dbx` and the name of your executable file.
  The prompt becomes: `(dbx)`.

  Example: Start `dbx` on the executable named `silly`:

```
demo% dbx silly
Reading symbolic information…
(dbx)
```

- To quit `dbx`, enter the `quit` command.

  Example: Quit `dbx`:

```
(dbx)quit                    (Skip this for now so you can do the next steps.)
demo%
```

- To set a breakpoint, wait for the dbx prompt, then type: stop in *subnam*, where *subnam* names a subroutine, function, or block data subprogram.

  Example: A way to stop at the first executable statement in a main program:

```
(dbx) stop in MAIN    { MAIN must be in uppercase. }
(2) stop in MAIN
(dbx)
```

  Although MAIN must be in uppercase, in general, *subnam* can be uppercase or lowercase. See “Case-Sensitive Variable Recognition (-U)” on page 190.

- To run the program from dbx, enter the run command, which runs the program in the executable files that were named when you started dbx.

  Example: Run the program from within dbx:

```
(dbx) run
Running: silly
stopped in MAIN at line 3 in file "a1.f"
   3      call mkidentity( twobytwo, n )
(dbx)
```

  When the breakpoint is reached, dbx displays a message showing where it stopped, in this case, at line 3 of the a1.f file.

- To print a value, enter the print command.

  Example: Print value of n:

```
(dbx) print n
n = 2
(dbx)
```

Example: Print the matrix `twobytwo`; the format may vary with the release:

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
(dbx)
```

Example: Print the matrix `array`:

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx)
```

The print fails because `array` is not defined here—only in `mkidentity`. The error message details may vary with the release, and, of course, with any translation.

- To advance execution to the next line, enter the `next` command.

Example: Advance execution to the next line:

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4      print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

The `next` command executes the current source line, then stops at the next line. It counts subprogram calls as single statements.

Compare `next` with `step`. The `step` command executes the next source line, or the next step into a subprogram, and so forth. In general, if the next executable source statement is a subroutine or function call, then:

- `step` sets a breakpoint at the first source statement of the subprogram.
- `next` sets the breakpoint at the first source statement after the call, but still in the calling program.

Segmentation Fault—Finding the Line Number

If a program gets a segmentation fault (`SIGSEGV`), it references a memory address outside of the memory available to it.

Some Causes of `SIGSEGV`

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.
- An array index is miscalculated.
- The calling routine calls has fewer arguments than required.
- A pointer is used before it is defined

Some Ways to Locate the Source Line

There are several ways to locate the offending source line. Any of the following ways can be helpful:

- Recompile with the `-Xlist` option to get global program checking.
- Recompile with `-C,` subscript checking option. See "Subscript Bounds (-C)."
- Use `dbx` to find the source code line where a segmentation fault occurred.

Example: Use a program to generate a segmentation fault:

```
demo 4% cat WhereSEGV.f
     INTEGER a(5)
     j = 2000000
     DO 9 i = 1,5
          a(j) = (i * 10)
9    CONTINUE
     PRINT *, a
     END
demo 5%
```

Example: Use `-C` to locate a segmentation fault:

```
demo 5% f77 -C -silent WhereSEGV.f
demo 6% a.out
Subscript out of range on file WhereSEGV.f, line 4, procedure
MAIN.
Attempt to access the 2000000-th element of variable a.
Abort (core dumped)
demo 7%
```

Example: Use `dbx` to find the line number of a segmentation fault:

```
demo 5% f77 -g -silent WhereSEGV.f
demo 6% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation fault (core dumped)
demo 7% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

Exceptions—Finding the Line Number

If a program gets an exception, there are many possible causes. One approach to locate the problem is to find the line number in the source program where the exception occurred, then look for clues there.

You can find the source code line number where a floating-point exception occurred by using the `ieee_handler` routine with either `dbx` or `debugger`.

Example: Find where an exception occurred:

WhereExcept.f

```
      EXTERNAL myhandler                                ! Main
      INTEGER ieeer, ieee_handler, myhandler
      REAL r/14.2/, s/0.0/
      ieeer = ieee_handler('set', 'all', myhandler)
      PRINT *, r/s
      END
      INTEGER FUNCTION myhandler(sig, code, context) ! Handler
*     {This handler is OK in SunOS 4.X/5.0 since it just aborts.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f77 -g -silent WhereExcept.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) catch FPE                               The catch FPE dbx command
(dbx) run
Running: a.out
signal FPE (floating point divide by zero)
      in MAIN at line 5 in file "WhereExcept.f"
    5          PRINT *, r/s
(dbx)
```

Bus Error—Finding the Line Number

If a program gets a bus error (`SIGBUS`), it usually has some problems with misaligned data. The address may well be valid, whereas with `SIGSEGV`, the address is invalid. Some possible causes of `SIGBUS` are:

- Misaligned data
- Using a pointer that is not defined or incorrectly defined

Example: Use a program to generate a bus error (`SIGBUS`):

```
demo% cat WhereSIGBUS.f
     character*1 c(5)
     call sub(c(2))  ! Assumes argument is aligned as a character,  bytes 2-5
     end
     subroutine sub(i) ! Assumes argument is aligned as an integer
     print *,i
     end
demo% f77 -C -silent WhereSIGBUS.f
demo% a.out
*** TERMINATING a.out
*** Received signal 10 (SIGBUS)
Bus Error (core dumped)
```

Example: Recompile with the `-Xlist` to locate a bus error (`SIGBUS`):

```
demo 5% f77 -Xlist -silent WhereSIGBUS.f
demo 6% cat WhereSIGBUS.lst
WhereSIGBUS.f             Fri Jun 10 16:02:17 1994                     page 1
FILE  "WhereSIGBUS.f"
     1          character*1 c(5)
     2          call sub(c(2))
                           ^
**** ERR  #418:  argument "c" is character, but dummy argument is integer*4
                 See: "WhereSIGBUS.f" line #4
     2          call sub(c(2))
                           ^
**** ERR  #316:  array "c" may be referenced before set by sub in line #5
     3          end
     4          subroutine sub(i)
     5          print *,i
 <many lines omited>
```

Trace of Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that brought it there. This sequence is called a *stack trace*.

Example: Show the sequence of calls, starting at where the execution stopped:

`ShowTrace.f` is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

Note the reverse order:

`MAIN` called `calc`
`calc` called `calcb`.

Execution stopped, line 23 →
`calcB` called from `calc`, line 9 →
`calc` called from `MAIN`, line 3 →

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
...
(dbx) run
Running: a.out
(process id 1089)
signal SEGV (no mapping at the fault address) in calcb at line 23
in file "ShowTrace.f"
   23                  v(j) = (i * 10)
(dbx) where
=>[1] calcb(v =  ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a =  ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
```

The `where` command shows where in the program flow execution stopped—how execution reached this point—that is, a *stack trace* of the called routines. Since you no longer get an *automatic* traceback, we have following ode.

Ode To Traceback

O blinding core! File of death!
Alone like Abel's brother, Seth.
The demise of process I cannot face
Without the aid of stackish trace.
To see what by you must needs be done,
Please see Example Twenty-One.[1]

© *Mateo Burtch, 1992*

1. Since `trace` be dead, or just not there, try `dbx`'s better `where`.
Seek not example twenty one, as it was cited just for fun.

Arrays

Example: dbx recognizes arrays and can print them:

Arraysdbx.f

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1         DIMENSION IARR(4,4)
    2         DO 90 I = 1,4
    3                 DO 20 J = 1,4
    4                         IARR(I,J) = (I*10) + J
    5   20            CONTINUE
    6   90    CONTINUE
    7         END
(dbx) stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7         END
(dbx) print IARR
iarr =
     (1,1) 11
     (2,1) 21
     (3,1) 31
     (4,1) 41
     (1,2) 12
     (2,2) 22
     (3,2) 32
     (4,2) 42
     (1,3) 13
     (2,3) 23
     (3,3) 33
     (4,3) 43
     (1,4) 14
     (2,4) 24
     (3,4) 34
     (4,4) 44
(dbx) print IARR(2,3)
     iarr(2, 3) = 23  ← Order of user-specified subscripts ok
(dbx) quit
demo%
```

Array Slices

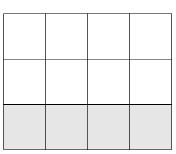
Example: `dbx` prints array *slices* if you specify which rows and columns:

`ShoSli.f`

This is one way of printing portions of large arrays.

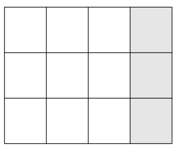
```
demo% f77 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
     1         INTEGER*4  a(3,4), col, row
     2         DO row = 1,3
     3                 DO col = 1,4
     4                   a(row,col) = (row*10) + col
     5                 END DO
     6         END DO
     7         DO row = 1, 3
     8                  WRITE(*,'(4I3)') (a(row,col),col=1,4)
     9         END DO
     10        END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
     7          DO row = 1, 3
(dbx)
```

Example: Print row 3:



```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)
```

Example: Print column 4:



```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

Intrinsic Functions

`dbx` recognizes FORTRAN 77 intrinsic functions.

Example: Show an intrinsic function in `dbx`:

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = 2
    END
demo% f77 -g -silent ShowIntrinsic.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10903)
stopped in MAIN at line 2 in file "ShowIntrinsic.f"
    2       i = 2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print abs(i)
abs(i) = 0
(dbx) quit
demo%
```

Complex Expressions

dbx also recognizes FORTRAN 77 complex expressions.

Example: Show a complex expression in dbx:

```
demo% cat ShowComplex.f
    COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f77 -g -silent ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

Logical Operators

dbx can locate FORTRAN 77 logical operators and print them.

Example: Show logical operators in dbx:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f77 -g -silent ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4  y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

Miscellaneous Tips

The following tips and background concepts can help. For more details, see the `dbx` documentation.

Current Procedure and File

During a debug session, `dbx` defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets one of three different breakpoints, depending on whether the current file is `a1.f, a2.f`, or `a3.f`.

Uppercase Letters

In general, if your program has uppercase letters in any identifiers, then the debugger recognizes them. You need not give it any specific case-sensitive or case-insensitive commands, as in some earlier versions.

`f77` and `dbx` must be in the same case-sensitive or case-insensitive mode:

- To compile and debug in case-insensitive mode, do so without the `-U` option. The debugger default then is: `dbxenv case insensitive`.

  If the source has a variable named `LAST`, then in dbx, both the `print LAST` or `print last` commands work. Both `f77` and `dbx` consider `LAST` and `last` to be the same, as requested.

- To compile and debug in case-sensitive mode, use `-U`. The debugger default is then `dbxenv case sensitive`.

  If the source has a variable named `LAST`, but one named `last`, then in dbx, `print LAST` works, but `print last` does *not* work. Both `f77` and `dbx` distinguish between `LAST` and `last`, as requested.

Note – File or directory names are always case-sensitive in both `debugger` and `dbx`. This rule is true even if you have set the `dbxenv case insensitive` environment attribute.

Optimized Programs

To debug optimized programs:

- Compile the main program with `-g` but with no `-O`*n*.
- Compile every other routine of the program with the appropriate `-O`*n*.
- Start the execution under `dbx`.
- Use `fix -g` *any*`.f` on the routine you want to debug, *but no* `-O`*n*.
- Use `continue` with that routine compiled.

Runtime Checking

The `dbx` *runtime checking* feature can be very helpful for standard C programs that use pointers, but not for standard FORTRAN 77 programs.

The more common FORTRAN 77 problem of an array index accessing outside of the array can be detected with `-C`; see “Subscript Bounds (-C)” on page 189.

Main Features of the Debugger

Be sure to read the Debugger manual for the following information:

- The full range of features in the debugger
- The window-based and mouse-based interface
- An appendix with more FORTRAN 77 examples

Overview of `dbx` Features Useful for FORTRAN 77

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

- *Fix* one routine, then *continue* executing without recompiling the others
- *Set watchpoints* to stop or trace if a specified item changes
- Solaris 2.x — *Collect data* for performance tuning
- *Graphically monitor* variables, structures, and arrays
- *Set breakpoints* (set places to halt in the program) at lines or in functions
- *Show values*—once halted, show or modify variables, arrays, structures, …
- *Step* through a program, one source or assembly line at a time
- *Trace* program flow—show sequence of calls taken
- *Invoke procedures* in the program being debugged

- *Step* over or into function calls; step up and out of a function call
- *Run, stop, and continue* execution at the next line or at some other line
- *Produce* `dbx`-safe I/O in the command window
- *Save and then replay* all or part of a debugging run
- *Stack*—examine the call stack, or move up and down the call stack
- *Program* scripts in the embedded Korn shell

Solaris 2.x
- *Follow programs* as they `fork`(2) and `exec`(2)

7.4 Debugging of Parallelized Code

The parallelization options limit the debugging capabilities of `dbx`.

If you compile a routine with `-g` and a parallelizing option, debugging with `dbx` is possible, and although you will not be able to print the value of variables, symbolic traceback is available with the `dbx where` command..

For solutions, see Section C.6, "Debugging Tips and Hints for Parallelized Code," on page 399.

7.5 Compiler Messages in Listing (`error`)

`error` is a utility program that inserts compiler diagnostics above the relevant line in the source file, as follows:

- The diagnostics include the standard compiler error and warning messages, but *not* the `-Xlist` error and warning messages.
- The diagnostics listing changes your source files.
- This function does not work if the source files are in a read-only directory.

`error`(1) is included in the operating system if it was installed with a developer install, rather than an end-user install; it is also included if you install the package, `SUNWbtool`. There is also a `man` page for `error`.

Method

The `error` utility associates compiler error diagnostics with the offending source lines. It recognizes and categorizes diagnostics from a variety of source-language processors, and inserts them as comments in the appropriate source file before the lines that caused the corresponding errors.

You can then read the source code along with its compiler diagnostics.

`error` *Utility*

Use `error` as follows (pass `stdout` and `stderr` from `f77` to `error`).

In `sh`:

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

In `csh`:

```
demo% f77 any.f |& error options
```

Options

The general form for using `error` with options is:

error [**-n**] [**-q**] [**-v**] [**-s**] [**-T**] [**-t** *suffixlist*] [**-S**] [*filename* **]**

- `-n`

  Do not change any files. This option sends all diagnostics to the standard output.

- `-q`

  Query before changing each file. If there is no `-q` option, then the compiler changes all the files it encounters during the compilation, except those files for discarded error messages.

- `-v`

  After all files have been changed, invoke `vi` to edit them, starting with the first one; then position the cursor at the first diagnostic. If `vi` cannot be located in the standard places, try `emacs`, `ex`, or `ed`.

- `-s`

  Print out statistics regarding error categorization.
- `-T`

  Produce a terse form of messages. This option is intended for standard output.
- `-t`*suffixlist*

  Change only files whose suffixes appear in *suffixlist*. *suffixlist* is a dot-separated list, and an asterisk (`*`) is acceptable as a wildcard.

  Example: Change only files with the suffixes, `.h`, `.f*`, or `.t`:

```
demo% error -t '.h.f*.t'
```

- `-S`

  Display the errors in the standard output as they are produced.
- *filename*

  Read error messages from *filename* rather than from the standard input.

Description

The `error` utility examines each line of its input and does the following:

- Determines the language processor that produced the message, the file name, and line number of the offending line.
- Inserts the message in the form of a special comment *into the source file* immediately preceding the erroneous line. It changes source files.

If the source line of a diagnostic cannot be determined, the diagnostic is sent to the standard output. The files remain unchanged.

Scanning with an Editor

The `error` utility inserts diagnostics in appropriate files after all input is read. The `-s` option allows previewing diagnostics before files are changed.

All diagnostics are inserted as one-line comments, starting with the marker ### and ending with %%%. These markers make it easy for a text processor to:

- Locate such messages in a file
- Remove such messages from a file

The line number of the offending line, along with the language processor that issued the message, appears in the comment line as well.

Redirecting and Piping

You can pass both standard output and standard error from `f77` to the `error` utility.

For example, in `sh`:

```
demo$ f77 myprog.f 2>&1 | error -q
```

In `csh`:

```
demo% f77 myprog.f |& error -q
```

In each shell, the command compiles and redirects or pipes the standard output and standard error to the `error` program. Then `error`, in turn, processes these diagnostics, and queries you before changing `myprog.f` and all other source files that are invoked from `myprog.f`.

Sample Use of `error`

Example: Sample program that shows how to use the `error` utility:

`forerror.f` (*before* compile)

This FORTRAN 77 source program contains various syntax errors.

Compile the program for `error` in `csh`:

```
demo% cat forerror.f
C    Sample program
C
     program test
     automatic x
     logical flag
     character*256 fname
     common /ioiflg/ ictl
     flag =.true.
     go to 10
     if (flag) then
10        ictl = 1
     else
          ictl = 0
     endif
     do 200 i = 0, MAXNUM
     call getenv(fname
     go to 200
     write (0, 2000) fname(:5)
200  continue
     endif
2000 format (’ This is a test ", b)
     end
demo% f77 -ansi forerror.f |& error
```

Example: Source file changed by the `error` utility:

`forerror.f` ***(**after* compile***)***

The source file has been changed.

```
demo% cat forerror.f
C###0 [Sunf77] ANSI extension: source line(s) in nonStandard
format%%%
C    Sample program
C
C###3 [Sunf77] ANSI extension: input contains lower case
letters%%%
     program test
C###4 [Sunf77] Warning: local variable "x" never used%%%
C###4 [Sunf77] ANSI extension: AUTOMATIC statement%%%
     automatic x
     logical flag
     character*256 fname
     common /ioiflg/ ictl
     flag =.true.
     go to 10
C###10 [Sunf77] Warning: statement cannot be reached%%%
     if (flag) then
C###11 [Sunf77] Warning: there is a branch to label 10 from outside
block%%%
10        ictl = 1
     else
          ictl = 0
     endif
     do 200 i = 0, MAXNUM
C###16 [Sunf77] Error: unclassifiable statement%%%
C###16 [Sunf77] Error: unbalanced parentheses, statement
skipped%%%
     call getenv(fname
     go to 200
C###18 [Sunf77] Warning: statement cannot be reached%%%
     write (0, 2000) fname(:5)
200  continue
C###20 [Sunf77] Error: endif out of place%%%
     endif
C###21 [Sunf77] Error: unclassifiable statement%%%
C###21 [Sunf77] Error: unbalanced quotes; closing quote
supplied%%%
C###21 [Sunf77] Error: unbalanced parentheses, statement
skipped%%%
2000 format (' This is a test ", b)
     end
```

Floating Point 8

This chapter is organized into the following sections.

This chapter introduces floating-point problems and IEEE floating-point tools for solving those problems.

If you are not familiar with floating-point arithmetic, see:

- The *Numerical Computation Guide*. which contains detailed explanations and examples
- The document, "*What Every Computer Scientist Should Know About Floating-point Arithmetic*," by David Goldberg. It can be found in the AnswerBook system or in the `READMEs` directory.

8.1 The General Problems

How can IEEE arithmetic help solve real problems? IEEE 754 standard floating-point arithmetic offers greater control over computation than is possible in any other type of floating point. In scientific research, there are many ways for errors to occur:

- The model may be wrong.
- The algorithm may be numerically unstable—solving equations by inverting $A^T A$, for example.
- The data may be ill-conditioned.
- The computer may be producing unexpected results.

It is nearly impossible to separate these error sources. Using library packages which have been approved by the numerical analysis community reduces the chance of there being a code error. Using good algorithms is another must. Using good computer arithmetic is the next obvious step.

The IEEE Standard represents the work of many of the best arithmetic specialists in the world today. It was influenced by the mistakes of the past. It is, by construction, better than the arithmetic employed in the S/360 family, the VAX family, the CDC, CRAY, and UNIVAC families, to name but a few. This is not because these vendors are not clever, but because the IEEE pundits came later and were able to evaluate the choices of the past and their consequences. Does IEEE arithmetic solve all problems? No. But in general, the IEEE Standard makes it easier to write better numerical computation programs.

8.2 IEEE Solutions

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

For rounding, IEEE arithmetic defaults to doing the intuitive thing, and closely corresponds with old arithmetic.

IEEE offers choices, which the expert can use to good effect, while old arithmetic did it just one way.

What happens if we:

- Multiply two very large numbers with the same sign?
- Have large numbers of different signs?
- Divide nonzero by zero?
- Divide zero by zero?

In old arithmetic, all these cases are the same. The program aborts on the spot; in some very old machines, the computation proceeds, but with garbage. IEEE provides choices.

The default solution is to produce the following:

```
big*big = +Inf
big*(-)big = -Inf
num/0.0 = +Inf   Where num > 0.0
num/0.0 = -Inf   Where num < 0.0
0.0/0.0 = NaN    Not a Number
```

In the above example `+Inf`, `-Inf`, and `NaN` are introduced intuitively. More details later.

Also, an exception of one of the following kinds is raised:

- *Invalid*—Examples that yield invalid are 0.0/0.0, sqrt(-1.0), log(-37.8), …
- *Division by zero*—Examples that yield division by zero are 9.9/0.0, …
- *Overflow*—Example with overflow: `MAXDOUBLE`+0.0000000000001e308
- *Underflow*—Example that yields underflow: `MINDOUBLE` * `MINDOUBLE`
- *Inexact*—Examples that yield inexact are 2.0 / 3.0, log(1.1), read in 0.1, …
  No exact representation in binary for the precision is involved.

There are various reasons why all this works is important:

- If you do not understand what you are using, you may not like the results.
- Poor arithmetic can produce poor results, which cannot be easily distinguished from other causes of poor results.
- Switching everything to double precision is no panacea.

8.3 IEEE Exceptions

IEEE exception handling is the default on a SPARC processor. However, there is a difference between *detecting* a floating-point exception, and *generating a signal* for a floating-point exception (SIGFPE).

Detecting a Floating-point Exception

In accordance with the IEEE Standard, two things happen when a floating-point exception occurs in the course of an operation.

- The handler returns a default result. For 0/0, return NaN as the result.
- A flag is set that an exception is raised. For 0/0, set "invalid operation" to 1.

Generating a Signal for a Floating-point Exception

The default on SPARC hardware systems is that they do *not* generate a *signal* for a floating-point exception. The assumption is that signals degrade performance, and that most developers do not care about most exceptions.

To generate a signal for a floating-point exception, you establish a signal handler. You use a predefined handler or write your own. See "Exception Handlers and ieee_handler()" on page 226.

Default Signal Handlers

By default, `f77` sets up some signal handlers, mostly for dealing with such things as a floating-point exception, interrupt, bus error, segmentation violation, or illegal instruction.

Although, generally, you would not want to turn off this default behavior, you can do so by setting the global C variable `f77_no_handlers` to 1, as shown in the following steps.

1. Create a C program.

```
demo% cat NoHandlers.c
   int  f77_no_handlers=1 ;
demo%
```

2. Compile it and save the .o file.

```
demo% cc -c -o NoHand NoHandlers.c
```

3. Link the corresponding .o file into your executable file.

```
demo% f77 NoHand.o Any.f
```

Otherwise, by default, it is 0. The effect is felt just before execution is transferred to the program, so it does not make sense to set or unset it there.

This variable is in the name space of the program, so do not use `f77_no_handlers` as the name of a variable anywhere else other than in the above C program.

8.4 IEEE Routines

The following interfaces help people use the functionality of IEEE arithmetic. These are mostly in the math library `libsunmath` and in several `.h` files.

- `ieee_flags`(3m)—Control rounding direction and rounding precision. Query exception status. Clear exception status.
- `ieee_handler`(3m)—Establish exception handler. Remove exception handler.
- `ieee_functions`(3m)—List name and purpose of each IEEE function.
- `ieee_values`(3m)—A list of functions that return special values.
- Other `libm` functions:
  - `ieee_retrospective`
  - `nonstandard_arithmetic`
  - `standard_arithmetic`

Many vendors support the IEEE Standard. The SPARC processors conform to the IEEE Standard in a combination of hardware and software support for different aspects.

The older Sun-4 uses the Weitek 1164/5, and the Sun-4/110 has that as an option.

The newer Sun-4 and the SPARC system series both use floating-point units with hardware square root. This is accessed if you compile with the `-cg89` option.

The newest SPARC system series uses new floating-point units, including SuperSPARC, with hardware integer multiply and divide instructions. These are accessed if you compile with the `-cg92` option.

The utility `fpversion` tells which floating-point hardware is installed. This utility runs on all Sun architectures. See `fpversion`(1), and read the *Numerical Computation Guide* for details. This utility replaces the older utility, `fpuversion4`.

Flags and `ieee_flags()`

The `ieee_flags` function is used to query and clear exception status flags. It is part of the `libsunmath` shipped with SPARC operating systems, and performs the following tasks.

- Control rounding direction and rounding precision
- Check the status of the exception flags
- Clear exception status flags

The general form of a call to `ieee_flags` is as follows:

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

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

Possible parameter values are shown in the following table:

```
action:    get, set, clear, clearall
mode:      direction, precision, exception
in,out:    nearest, tozero, negative, positive,
           extended, double, single,
           inexact, division, underflow, overflow, invalid,
           all, common
```

The meanings of the possible values for `in` and `out` depend on the action and mode they are used with. These are summarized in the following table.

Table 8-1 `ieee_flags` Argument Meanings

Value of *in* and *out*	**Refers to**
`nearest`, `tozero`, `negative`, `positive`	Rounding direction
`extended`, `double`, `single`	Rounding precision
inexact, division, underflow, overflow, invalid	Exceptions
`all`	All 5 exceptions
`common`	Common exceptions: invalid, division, overflow

Note – These examples show only how to call the routines to get the information or set the behavior. They make no attempt to teach the numerical analysis that lets you know when to call them or what behavior to set.

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

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

Also, to determine if the `overflow` exception flag is raised, set the input argument `in` to `overflow`. On return, if *out* equals `overflow`, then the `overflow` exception flag is raised; otherwise it is not raised.

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

Example: Clear the `invalid` exception:

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

Example: Clear all exceptions:

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

Example: Set rounding direction to zero:

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

Example: Set rounding precision to `double`:

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

Turning Off All Warning Messages with `ieee_flags`

Use this option if you do not want to know about the unrequited exceptions. To do this, clear all accrued exceptions by putting a call to `ieee_flags()` just before your program exits.

Example: Clear all accrued exceptions with `ieee_flags()`:

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

Detecting an Exception with `ieee_flags`

These examples show only how to call the routines to get the information. They make no attempt to teach the numerical analysis that lets you know when to call them and what to do with the information.

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

(Solaris 2.x)
`DetExcFlg.F`

```
#include "f77_floatingpoint.h"
      CHARACTER*16 out
      DOUBLE PRECISION d_max_subnormal, x
      INTEGER div, flgs, inv, inx, over, under

        x = d_max_subnormal() / 2.0               ! Cause underflow

        flgs=ieee_flags('get','exception','',out) ! Which are raised?

        inx   = and(rshift(flgs, fp_inexact)  , 1) ! Decode
        div   = and(rshift(flgs, fp_division) , 1)  ! the value
        under = and(rshift(flgs, fp_underflow), 1)   ! returned
        over  = and(rshift(flgs, fp_overflow) , 1)    ! by
        inv   = and(rshift(flgs, fp_invalid)  , 1)     ! ieee_flags

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

Use the `.F` suffix so the preprocessor brings in the `f77_floating.h` header file.

Example: Compile and run to detect an exception with `ieee_flags`:

```
demo% f77  -silent DetExcFlg.F
demo% a.out
 Highest priority exception is: underflow
  invalid  divide  overflo underflo inexact
       0       0       0       1       1
 (1 = exception is raised; 0 = it is not)
demo%
```

Detecting All Five Exceptions with `ieee_flags`

How to call, not when to call or what to do with the information:

Example: Detect all five exceptions using `ieee_flags`, and decode them:

DetAllFlg.F

```
#include "f77_floatingpoint.h"
    CHARACTER*16 out
    DOUBLE PRECISION d_max_normal, d_max_subnormal, x, y /0.0/
    INTEGER div, flgs, inv, inx, over, under

    x = log( -37.8 )                          ! Cause invalid
    x = 3.14159 / y                           ! Cause division by zero
    x = d_max_subnormal() / 2.0               ! Cause underflow
    x = d_max_normal() * 2.0D0                ! Cause overflow
    x = 2.0D0 / 3.0D0                         ! Cause inexact

    flgs=ieee_flags('get','exception','',out)!which exceptions raised?

    inx   = and(rshift(flgs, fp_inexact)  , 1)  ! Decode the
    div   = and(rshift(flgs, fp_division) , 1)   ! value
    under = and(rshift(flgs, fp_underflow), 1)    ! returned in
    over  = and(rshift(flgs, fp_overflow) , 1)     ! flgs, using
    inv   = and(rshift(flgs, fp_invalid)  , 1)      ! bit-shifts

    PRINT *, "Highest priority exception is: ", out

    PRINT *, ' invalid  divide  overflo underflo inexact' ! 1=raised
    PRINT '(5i8)', inv,div,over,under,inx              ! 0=not raised
    i = ieee_flags('clear', 'exception', 'all', out)! Clear all
    END
```

Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

Compile and run to detect all five exceptions with `ieee_flags`:

```
demo% f77  -silent DetAllFlg.F
demo% a.out
Highest priority exception is: invalid
 invalid  divide  overflo underflo inexact
       1       1       1       1       1
demo%
```

Values and `ieee_values`

The `ieee_values`(3m) file describes a collection of functions. Each function returns a special IEEE value. You can use these special IEEE entities, such as *infinity* or *minimum normal*, in a user program.

Example: A convergence test may be like this:

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

The values available are listed in the following table.

Table 8-2 Functions for Using IEEE Values

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

For the two `NaN` functions, you can assign or print out the values, but comparisons using either of them always yield false. To determine whether some value is a `NaN`, use the function `ir_isnan(r)` or `id_isnan(d)`.

The FORTRAN 77 names for these functions are listed in:

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

Also see:

- `ieee_values`(3m)
- The `f77_floatingpoint.h` header file

Exception Handlers and `ieee_handler()`

Most floating-point users need to know the following about IEEE exceptions:

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

To obtain this information, you need to generate a signal for a floating-point exception. The official UNIX name for *signal: floating-point exception* is `SIGFPE`. To generate a `SIGFPE`, establish a signal handler. The default on SPARC hardware systems is that they do *not* generate a `SIGFPE`.

Establishing a Signal Handler Function with `ieee_handler()`

To establish a function as a signal handler, pass the name of the function to `ieee_handler()`, together with the exception to watch for and the action to take. Once you establish a handler, a signal is generated whenever the particular floating-point exception occurs.

The form of invoking `ieee_handler()` is:

`i = ieee_handler(` *action*`,` *exception*`,` *handler* `)`		
action	`character`	`get`, `set`, or `clear`
exception	`character`	`invalid`, `division`, `overflow`, `underflow`, or `inexact`
handler	function name	The name of the function you wrote, or `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, or `SIGFPE_ABORT`
return value	`integer`	`0=OK`

There are two general kinds of signal handler functions:

- Predefined signal handler functions
- Functions that you write yourself

Writing Predefined Signal Handler Functions

The predefined handlers are:

- `SIGFPE_DEFAULT` (*better to get default behavior without calling this*)
- `SIGFPE_IGNORE`
- `SIGFPE_ABORT`

Actions taken by the function are up to you. However, the function must be an integer function and must have three arguments and data types, as follows:

Solaris 2.x

- `hand5x( sig, sip, uap )`
  - `hand5x` is the name for your integer function.
  - `sig` is an integer.
  - `sip` is a record which has the structure `siginfo` (see example below).
  - `uap` is not used here.

Example: Form of signal handler function, *Solaris 2.x*:

```
INTEGER FUNCTION hand( sig, sip, uap ) ! Form, Handler, Solaris 2.x
INTEGER sig, location
STRUCTURE /fault_typ/
   INTEGER address
END STRUCTURE
STRUCTURE /siginfo/
   INTEGER si_signo
   INTEGER si_code
   INTEGER si_errno
   RECORD /fault_typ/ fault
END STRUCTURE
RECORD /siginfo/ sip
location = sip.fault.address
... actions you take ...
END
```

If the handler installed by `ieee_handler()` is written in FORTRAN 77, then the handler should not make any reference to the first argument (`sig` in the example above). The first argument is passed by value, but is expected by reference in a FORTRAN 77 handler. The actual signal number can be referenced as `loc(sig)`.

Solaris 1.x

- `hand4x( sig, code, context )`
  - `hand4x` is the name for your integer function.
  - `sig` is an integer.
  - `code` is an integer.
  - `context` is an array of five integers.

  Example: Form of signal handler function (*Solaris 1.x)*:

```
    INTEGER FUNCTION hand( sig, code, context ) ! Form, Handler, 1.x
    INTEGER sig, code, context(5)
    location = context(4)
    ... actions you take ...
    END
```

Detecting an Exception by Handler (Solaris 2.x and 1.x)

These examples show only how to call the routines for the information. They make no attempt to teach the numerical analysis that lets you know when to call them and what to do with the information.

Example: Detect exception, by handler (*Solaris 2.x and 1.x*):

Solaris 1.x/ 2.x
`DetExcHan.f`

```
      EXTERNAL myhandler                          ! Main
      REAL r / 14.2 /, s / 0.0 /
      i = ieee_handler ('set', 'division', myhandler )
      t = r/s
      END

      INTEGER FUNCTION myhandler(sig,code,context)! Handler, 2.x or 1.x
*       {OK in Solaris 2.x/1.x since all it does is abort.}
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f77 -silent DetExcHan.f
demo%  a.out
abort: called
Abort (core dumped)
demo%
```

`SIGFPE` is generated whenever that floating-point exception occurs. Then the `SIGFPE` is detected, and control is passed to the `myhandler` function.

Locating an Exception by Handler (Solaris 2.x)

Example: Locate an exception (get *address*) using a handler (*Solaris 2.x*):

Solaris 2.x
`LocExcHan5x.F`

```
#include "f77_floatingpoint.h"
    EXTERNAL hand5x                              ! Main
    INTEGER hand5x, i, ieee_handler
    REAL r / 14.2 /,  s / 0.0 /, t
    i = ieee_handler( 'set', 'division', hand5x )
    t = r/s
    END

    INTEGER FUNCTION hand5x( sig, sip, uap)! Handler, Solaris 2.x
    INTEGER sig, location
    STRUCTURE /fault_typ/
        INTEGER address
    END STRUCTURE
    STRUCTURE /siginfo/
        INTEGER si_signo
        INTEGER si_code
        INTEGER si_errno
        RECORD /fault_typ/ fault
    END STRUCTURE
    RECORD /siginfo/ sip
    location = sip.fault.address
    WRITE (*,10)  location          ! Caveat: I/O in a handler is risky.
10  FORMAT('Exception at hex address ', Z8 )
    CALL abort()                    ! This reduces the risk mentioned above.
    END
demo%
```

Solaris 2.x

```
demo% f77 -silent LocExcHan5x.F
demo%  a.out
Exception at hex address    10DC4    {The actual address varies with }
abort: called                        {installation and architecture.}
Abort (core dumped)
demo%
```

Note – An *address* is mostly for those who use such low-level debuggers as adb.

Locating an Exception by Handler (Solaris 1.x)

Example: Locate an exception (get address) using a handler (*Solaris 1.x)*:

Solaris 1.x
`LocExcHan4x.f`

```
      EXTERNAL hand4x                              !  Main
      INTEGER hand4x, i, ieee_handler
      REAL r /14.2/,  s /0.0/, t
      i = ieee_handler('set', 'division', hand4x)
      t = r / s
      END

      INTEGER FUNCTION hand4x(sig,code,context) ! Handler, Solaris 1.x
      INTEGER sig, code, context(5)
      WRITE( *, '("Exception at pc", I5 )' ) context(4)
      CALL abort()                                 !   Just to reduce risk
      RETURN
      END
```

Caveat: Above, I/O in a handler is risky.

Solaris 1.x

```
demo% f77 -silent LocExcHan4x.f
demo%  a.out
Exception at pc 8980
abort: called
Abort
demo%
```

Above, the actual address varies with installation and architecture.

Note – How to call, not when to call or what to do with the information.

Detecting All Exceptions by Handler (Solaris 2.x)

Example: Detect and locate all exceptions with a signal handler (*Solaris 2.x*):

Solaris 2.x
`DetAllHan5x.F`

```
#include "f77_floatingpoint.h"
  DOUBLE PRECISION x, y, d_max_normal, d_min_normal, z/0.0d0/
  EXTERNAL continue5x
  INTEGER continue5x
  ieeer = ieee_handler('set', 'all', continue5x) ! Establish handler
  IF (ieeer.ne.0) PRINT *,'cannot establish handler: continue5x'
ieeer = ieee_handler('set', 'inexact', SIGFPE_IGNORE) ! Ignore inexact

  WRITE(*,"(/'0/0:')")
  x = 0.0d0 / z                                         ! Invalid
  WRITE(*,"(/'3.14159/0.0 Trapped:')")
  x = 3.14159d0 / z                                     ! Div by 0, trapped

  WRITE(*,"(/'max_normal**2:')")
  y = d_max_normal()
  x = y * y                                             ! Overflow
  WRITE(*,"(/'min_normal**2:')")
  y = d_min_normal()
  x = y * y                                             ! Underflow

  ieeer = ieee_handler('set', 'inexact', continue5x)! Trap inexact
  IF (ieeer.ne.0) PRINT *,"Can't set inexact, handler continue5x"
  WRITE(*,"(/'2.0/3.0:')")
  x = 2.0d0 / 3.0d0                                      ! Inexact

ieeer = ieee_handler('clear','division',SIGFPE_DEFAULT)! Set div dflt
  IF (ieeer.ne.0) PRINT *, 'cannot clear division handler'
  WRITE(*,"(/'3.14159/0.0 Untrapped:')")
  x = 3.14159d0 / z                                    ! Div by 0, untrapped
  WRITE(*,"(' 3.14159/0.0 = ', F12.8/)") x
  END
...Continued...
```

Use the `.F` suffix so the preprocessor will bring in the `f77_floating.h` header file.

Note – How to call, not when to call or what to do with the information.

Example: Detect and locate all exceptions with handler (*Solaris 2.x*) (*Continued*).

Solaris 2.x
DetAllHan5x.F (*Continued*)

Handler →

< Uses sysmachsig.h values > which is schlepped in by f77_floatingpoint.h.

These codes may be different on Solaris x86. Check the /usr/include/sys/machsig.h file.

```
    INTEGER FUNCTION continue5x(sig, sip, uap) ! Handler-2.x
    INTEGER sig, code, location
    CHARACTER*9 label
    STRUCTURE /fault_typ/
       INTEGER address
    END STRUCTURE
    STRUCTURE /siginfo/          ! Translate siginfo_t in sys/siginfo.h
       INTEGER si_signo
       INTEGER si_code
       INTEGER si_errno
       RECORD /fault_typ/ fault
    END STRUCTURE
    RECORD /siginfo/ sip
    code = sip.si_code     ! Which exception raised SIGFPE?
    IF (code .eq. 3) label = 'division' ! These
    IF (code .eq. 4) label = 'overflow'  ! 5 codes
    IF (code .eq. 5) label = 'underflow'  ! are defined
    IF (code .eq. 6) label = 'inexact'     ! in
    IF (code .eq. 7) label = 'invalid'      ! sys/machsig.h
    location = sip.fault.address
    WRITE(*,10) label, code, location   ! I/O in handler is risky
10  FORMAT(A10,' exception, sigfpe code',I2,',at address ',Z8)
    END
```

Solaris 2.x
Compile, load, and run.

The addresses vary, depending on installation and architecture. The addresses are mostly for those who use such low-level debuggers as adb. See page 237 on how to get the source line number.

```
demo% f77 -silent DetAllHan5x.F
demo% a.out
0/0:
 invalid   exception, sigfpe code 7, at address    11144
3.14159/0.0 Trapped:
 division  exception, sigfpe code 3, at address    11190
 max_normal**2:
 overflow  exception, sigfpe code 4, at address    111DC
min_normal**2:
 underflow exception, sigfpe code 5, at address    11228
2.0/3.0:
 inexact   exception, sigfpe code 6, at address    1130C
3.14159/0.0 Untrapped:
 3.14159/0.0 = Infinity
 ... retrospective messages about exceptions ...
demo%
```

Example: Detect and locate all exceptions with a signal handler (*Solaris 1.x*):

Solaris 1.x
`DetAllHan4x.F`

Main→

Use the .F suffix so the preprocessor will bring in the f77_floating.h header file.

The next page has details on `inexact`. →

Handler→

Note special codes.

```
#include "f77_floatingpoint.h"
     DOUBLE PRECISION x, y, d_max_normal, d_min_normal, z/0.0d0/
     EXTERNAL continue4x
     INTEGER continue4x
     ieeer = ieee_handler('set', 'all', continue4x)    ! Establish handler
   IF (ieeer.ne.0) PRINT *,'cannot establish handler: continue4x'
     ieeer = ieee_handler('set','inexact',SIGFPE_IGNORE)! Ignore inexact

     WRITE(*,"(/'0/0:')")
     x = 0.0d0 / z                                        !  Invalid
     WRITE(*,"(/'3.14159/0.0:')")
      x = 3.14159d0 / z                                  !  Div by 0, trapped

     WRITE(*,"(/'max_normal**2:')")
     y = d_max_normal()
     x = y * y                                            !  Overflow
     WRITE(*,"(/'min_normal**2:')")
     y = d_min_normal()
     x = y * y                                            !  Underflow

     ieeer = ieee_handler('set', 'inexact', continue4x)! Trap inexact
     IF (ieeer.ne.0) PRINT *,'Cannot establish handler: inexact'
     WRITE(*,"(/'2.0/3.0:')")
     x = 2.0d0 / 3.0d0                                    !  Inexact

     ieeer=ieee_handler('clear','division',SIGFPE_DEFAULT)! Div default
     IF (ieeer.ne.0) PRINT *, 'could not clear division handler'
     WRITE(*,"(/'3.14159/0.0:')")
     x = 3.14159d0 / z                                 !  Div by 0, untrapped
     WRITE(*,"(' 3.14159/0.0 = ', F12.8/)") x
     END

     INTEGER FUNCTION continue4x(sig,code,sigcontext) !  Handler- 1.x
     INTEGER code, sig, sigcontext(5)
     CHARACTER label*16
     IF (loc(code) .eq. 208) label = 'invalid'
     IF (loc(code) .eq. 200) label = 'division by zero'
     IF (loc(code) .eq. 212) label = 'overflow'
     IF (loc(code) .eq. 204) label = 'underflow'
     IF (loc(code) .eq. 196) label = 'inexact'
     WRITE (*,1) loc(code), label, sigcontext(4) ! I/O in handler is risky
 1   FORMAT(' ieee exception code',I4, ',', A17, ',', ' at pc',I6)
     END
```

Example: Detect and locate all exceptions, with handler (*Continued*):

Solaris 1.x
Compile, load, and run.

The addresses vary, depending on installation and architecture. The addresses are mostly for those who use such low-level debuggers as adb.

See page 237 on how to get the source line number.

```
demo% f77 -silent DetAllHan4x.F
demo% a.out
0/0:
 ieee exception code 208, invalid         , at pc  9176

3.14159/0.0 Trapped:
 ieee exception code 200, division by zero, at pc  9252

max_normal**2:
 ieee exception code 212, overflow        , at pc  9328

min_normal**2:
 ieee exception code 204, underflow       , at pc  9404

2.0/3.0:
 ieee exception code 196, inexact         , at pc  9632

3.14159/0.0 Untrapped:
 3.14159/0.0 = Infinity

 Note: the following IEEE floating-point arithmetic exceptions
 occurred and were never cleared; see ieee_flags(3M):
 Division by Zero;
 Note: IEEE Infinities were written to ASCII strings or output files;
see econvert(3).
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Inexact;  Underflow;  Overflow;  Invalid Operand;
 Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
demo%
```

In the above example, after the execution of `x=2.0d0/3.0d0`, `x` contains:

x contains	**Solaris 2.x**	**Solaris 1.1.3 and Later**	**Before Solaris 1.1.3**
Untrapped `inexact`	0.666…	0.666…	0.666…
Trapped `inexact`	garbage	garbage	0.666…

The value is “garbage” because it is unpredictable; it depends on various actions that happen immediately before the exception.

Retrospective

The `ieee_retrospective` function queries the floating-point status registers to find out which exceptions have accrued. If any exception has a raised accrued exception flag, a message is printed to standard error to inform the programmer which exceptions were raised but not cleared. For FORTRAN 77, this function is called automatically just before normal termination. The message typically looks like this; the format varies with each release:

```
NOTE: The following IEEE floating-point arithmetic exceptions
occurred and were never cleared: Inexact; Division by Zero;
Underflow; Overflow; Invalid Operand. Sun's implementation of
IEEE arithmetic is discussed in the Numerical Computation Guide.
```

Nonstandard Arithmetic

Another useful math library function is *nonstandard arithmetic*.

The IEEE Standard for arithmetic specifies a way of handling underflowed results gradually by dynamically adjusting the radix point of the significand. Recall that in IEEE floating-point format, the radix point occurs before the significand, and there is an implicit leading bit of 1. Gradual underflow allows the implicit leading bit to be cleared to 0 and to shift the radix point into the significand, when the result of a floating-point computation would otherwise underflow. This result is not accomplished in hardware on a SPARC processor, but in software. If your program happens to generate many underflows (perhaps a sign of a problem with your algorithm?), and you run on a SPARC processor, you may experience a performance loss.

To turn off gradual underflow, compile with `-fnonstd`, or insert this line:

```
    CALL nonstandard_arithmetic()
```

To turn on gradual underflow (after you have turned it off), insert this line:

```
    CALL standard_arithmetic()
```

Legacy

- The `standard_arithmetic()` subroutine corresponds exactly to an earlier version named `gradual_underflow()`.
- The `nonstandard_arithmetic()` subroutine corresponds exactly to an earlier version named `abrupt_underflow()`.

Messages about Floating-point Exceptions

For FORTRAN 77, the current default is to display a list of accrued floating-point exceptions at the end of execution. In general, you get a message if any one of the invalid, division-by-zero, or overflow exceptions occur. Since most real programs raise inexact exceptions, you get a message if exceptions other than inexact exceptions occur. If it is only inexact, then no message is issued.

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

In your own exception handler routine, you can:

- Specify actions
- Turn off messages with `ieee_flags()` by clearing exception status flags

---

Note – Clearing all messages is not recommended. If you need to turn off these messages, record invalid, division-by-zero, and overflow some place.

8.5 Debugging IEEE Exceptions

You may want to debug programs that generate messages like this:

```
NOTE: the following IEEE floating-point arithmetic exceptions
occurred and were never cleared: Inexact; Division by Zero;
Underflow; Overflow; Invalid Operand. Sun's implementation of
IEEE arithmetic is discussed in the Numerical Computation Guide.
```

To locate the *line number* where the exception occurred, do the following:

- Establish a signal handler so that a `SIGFPE` is generated.
- After you invoke `dbx`, enter the `catch FPE` command.

Locating such a line number is shown in the following example. Also see page 226 for details about exception handlers.

You can find the source code line where a floating-point exception occurred by using the `ieee_handler` routine with either `dbx` or `debugger`.

Example: Locate the *line number* of an exception, `dbx` / handler (2.x and 1.x):

Solaris 2.x and 1.x
`LocExcDbx.f`

```
demo% cat LocExcDbx.f
    INTEGER myhandler                                     !  Main
    EXTERNAL myhandler
    REAL r /14.2/, s /0.0/
    ieeer = ieee_handler('set', 'common', myhandler)
    PRINT *, r/s
    END

    INTEGER FUNCTION myhandler( sig, code, context ) ! Handler
!     {OK in Solaris 2.x/1.x, since all it does is abort.}
    INTEGER sig, code, context(5)
    CALL abort()
    END
demo% f77 -g -silent LocExcDbx.f
demo% dbx a.out
Reading symbolic information …
(dbx) catch FPE                          {Note the catch FPE  command.}
(dbx) run
Running: a.out
signal FPE (floating point exception)
           in MAIN at line 5 in file "LocExcDbx.f"
    5   PRINT *, r/s
(dbx) quit
demo%
```

8.6 Guidelines

To sum up, SPARC arithmetic is a state-of-the art implementation of IEEE arithmetic, optimized for the most common cases.

More problems can safely be solved in single precision, due to the clever design of IEEE arithmetic.

To get the benefits of IEEE math for most applications, if your program gets one of the common exceptions, then you probably want to continue with a sensible result. That is, you do *not* want to use `ieee_handler` to *abort* on the common exceptions.

If your system time is very large with over 50% of runtime, look into modifying your code or using `nonstandard_arithmetic`.

8.7 Miscellaneous Examples

A miscellaneous collection of examples is provided here as additional tips.

Kinds of Problems

The problems in this chapter usually involve arithmetic operations with a result of invalid, division by zero, overflow, underflow, or inexact.

For instance, take underflow—in *old* arithmetic, that is, prior to IEEE, if you multiply two very small numbers on a computer, you get zero. Most mainframes and minicomputers behave that way. In IEEE arithmetic, there is *gradual underflow*, which expands the dynamic range of computations.

For example, consider a machine with `1.0E-38` as the machine *epsilon*, the smallest representable value on the machine. Multiply two small numbers.

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

In old arithmetic, you get `0.0`, but with IEEE arithmetic and the same word length, you get `1.40130E-45`. With old arithmetic, if a result is near zero, it becomes zero. This result can cause problems, especially when you are subtracting two numbers, because this is a principal way accuracy is lost.

You can also detect that the answer is inexact. The `inexact` exception is common, and means the calculated result cannot be represented exactly, at least not in the precision being used, but it is as good as can be delivered.

Underflow tells us, as we can tell in this case, that we have an answer smaller than the machine naturally represents. This result is accomplished by "stealing" some bits from the mantissa and shifting them over to the exponent. The result is less precise, in some sense, but more so in another. The deep implications are beyond this discussion. If you are interested, consult *Computer*, January 1980, Volume 13, Number 1, particularly I. Coonen's article, "*Underflow and the Denormalized Numbers*."

Most scientific programs have sections of code that are sensitive to roundoff, often in an equation solution or matrix factorization. So be concerned about numerical accuracy—if your computer doesn't do a good job, your results will be tainted, and there is often no way to know that this has happened.

Simple Underflow

Some applications actually do a lot of work very near zero. This is common in algorithms which are computing residuals or differential corrections. For maximum numerically safe performance, perform the key computations in extended precision. If the application is a single-precision application, this is easy, as we can perform key computations in double precision.

Example: A simple dot product computation:

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

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

```
    REAL*8 sum
    DO i = 1, n
       sum = sum + dble(a(i)) * dble(b(i))
    END DO
    result = sum
```

It may be advisable to have both versions, and switch to the double precision version only when required.

You can force a SPARC processor to behave like an older computer with respect to underflow. Add the following line to your FORTRAN 77 main program:

```
    CALL nonstandard_arithmetic()
```

Bee aware, however, that you are giving up the numerical safety belt that is the operating system default. You can get your answers faster, and you won't be any less safe than, say, a VAX, but use at your own risk.

Continuing with Wrong Answer

You might wonder why continue if the answer is clearly wrong. The general idea is that IEEE arithmetic allows you to make distinctions about what kind of *wrong*, such as NaN or Inf. Then decisions can be made based on such distinctions.

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

It is possible to carefully arrange each part of the calculation to coerce each subresult to the correct range.

```
 4.0  < computed <  Inf →  5 volts
-4.0  ≤ computed ≤  4.0 →  0 volts
-Inf  < computed ≤ -4.0 → -5 volts
```

Furthermore, since `Inf` is not an allowed value, you need special logic to ensure that big numbers are not multiplied.

IEEE arithmetic allows the logic to be much simpler, as the computation can be written in the obvious fashion, and only the final result need be coerced to the correct value, since ±`Inf` can occur, and can be easily tested.

Furthermore, the special case of 0/0 can be detected and dealt with as you wish. The result is easier to read, and faster in executing, since you don’t do unneeded comparisons.

Excessive Underflow

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

For some SPARC platforms, the hardware, being designed for the typical case, does *not* produce a result; instead, software is employed to compute the correct IEEE complying result. As you may guess, this method is much slower. In the majority of applications, it is invisible. When it is not, the symptom is that the system time component of your runtime, which can be determined by running your application with the `time` command, is much too large.

For other SPARC platforms, the hardware does produce the result at a much faster speed.

The following examples have varying differences, depending on the platform.

Example: Excessive underflow:

DotProd.f

```
      PROGRAM dotprod
      INTEGER maxn
      PARAMETER (maxn=10000)
      REAL a(maxn), b(maxn), eps /1.0e-37/, sum
      DO i = 1, maxn
          a(i) = 1.0e-30
          b(i) = 1.0e-15
      END DO
      sum = 0.
      DO i = 1, maxn
          sum = sum + a(i)*b(i)
      END DO
      END
```

After compiling and running `dotprod`, the results of the `time` command are:

```
        2.3 real          0.1 user          1.8 sys
```

The real computation took about 0.1 second, but the software fix took two seconds. In a real application, the difference can be hours, and is not desirable, of course.

Solution 1: Change All of the Program

If you rewrite with all double precision, there is vast improvement in speed:

```
        0.2 real          0.0 user          0.1 sys
```

It may not be desirable to promote an entire program to double precision, though this is what is traditionally done to make up for the fact that old-style arithmetic is less accurate.

Solution 2: Change One Double Precision Variable

Declare only sum to be double precision, and change only the summation line of code as follows:

```
    sum = sum + a(i)*dble(b(i))
```

Doing so minimizes the software underflow problem:

```
        0.3 real          0.1 user          0.0 sys
```

In a real application, you should put the variable sum in double precision, and coerce it to single precision only on output. This is not a performance issue, but a numeric one. Of course, it may not be easy to tell which variables in a complex program need to be promoted. The effort is worthwhile, not only because of the performance (which, as you will learn, can be achieved in other ways), but because the numerics are enhanced as well.

Solution 3: Nonstandard Arithmetic

There is a “quick and dirty” solution, which is:

```
    CALL nonstandard_arithmetic()
```

This code tells the hardware to act like an old-style computer, and when underflow occurs, just flush to zero. A runtime results:

```
        0.5 real          0.0 user          0.1 sys
```

This time is about the same as promoting one variable to double. The difference is that now the computed result is 0. This is a bad result because if this dot product is really the final result, there is probably nothing wrong with this solution.

If, however, this result feeds into more elaborate computations, you have thrown away some information, which may be important. If the algorithm is stable, the input well conditioned, and the implementation careful, it does not matter. If there is anything else “shaky,” this result may push it over.

Solution 4: The `-r8` *Option*

Another quick fix is to use the `-r8` option. This workaround is safe, but just a bit costly. It informs the compiler to interpret `REAL` as `DOUBLE PRECISION`. You may prefer this solution if the code was developed on a CRAY, CDC, or other 64-bit machine. In many cases, `-r8` suffices to produce correct results, thanks to the miracles of modern arithmetic, and is faster.

If you recompile `DotProd.f` with `-r8`, the `time` command results in:

```
        0.8 real         0.0 user         0.1 sys
```

If you wish to look further, read the section on `ieee_handler` and employ it to track down the affected lines.

`-r8` ***with Migrating***

Those migrating from chips like 68881 or 80387 processors may wonder why `-r8` is necessary. The code worked well (full speed) on their last machine. The reason is that these numeric processors provide internal registers which are 80-bit wide.

- **Advantage**

  An 80-bit FPU has the advantage that when everything fits in the 80-bit registers, the results are a little better.

- **Disadvantages**
  - An 80-bit FPU is typically slower or more expensive than either a 32-bit or a 64-bit FPU.
  - Since some intermediate results are computed with 80-bit precision, and others with only 32-bit or 64-bit precision, answers depend on exactly how the code is written, what optimization level is selected, the compiler version, and other factors not under your control. Results tend to vary, making it harder to validate the software, and so forth.

At this point, every SPARC processor performs arithmetic with 32-bit or 64-bit precision as coded by you.

If you are porting codes that were developed on old arithmetic machines, it is probably preferable to *stop* on overflows, division by zero, and so on. A solution is to use the `ieee_handler`, as in the examples.

The `-dalign` *Option*

If `–r8` is combined with `–dalign`, the program runs more slowly than without the `-r8` option. This is likely to happen if the key computational loops are very heavily exercised and involve mixed precision (double + single).

`-r8` *with Double Precision*

If `–r8` is used, and the key computational loops are very heavily exercised and involve double precision, then on SPARC platforms, the program runs more slowly than without the `-r8` option. The double precision is converted to quadruple precision, which is slower.

Porting from Other FORTRAN 77s 9

This chapter is organized into the following sections:

This chapter introduces porting programs from other dialects of FORTRAN 77. If you have VMS FORTRAN 77 programs, most compile almost exactly as is; if they don't, see the chapter on VMS extensions in the *FORTRAN 77 4.0 Reference Manual*.

9.1 General Hints

Keep these conventions in mind when transporting from another machine:

- Your source file name must have a `.f`, `.F`, or `.for` extension.
- If you are entering programs manually instead of reading them from tape, start lines with a tab or space so the code begins after column five, except for comments and labels.

9.2 Time Functions

When porting programs from a different FORTRAN 77 system, check the code to make sure that time functions used in the programs operate like those in this FORTRAN 77 compiler. If they do not, change the program to use equivalent functions.

The following time functions, which are found on some other machines, are not directly supported, but you can write subroutines to duplicate their functions:

- Time-of-day in `10h` format
- Date in `A10` format
- Milliseconds of job CPU time
- Julian date in ASCII

For example, to find the current Julian date, call `TIME()` to get the number of seconds since January 1, 1970, convert the result to days (divide by 86,400), and add 2,440,587 (the Julian date of December 31, 1969).

Several time functions are supported in the `f77` extensions to standard FORTRAN 77, and are described in the following two tables.

Table 9-1 Time Functions Available to FORTRAN 77

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

The routines listed in Table 9-2 provide compatibility with VMS FORTRAN 77 system routines. To use these routines, you must include the `-lV77` option on the `f77` command line, in which case you also get the VMS versions of `idate` and `time` instead of the standard versions.

Example: Using the `-lV77` option:

```
demo% f77 myprog.f -lV77
```

Table 9-2 Summary: VMS FORTRAN 77 System Routines

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

The error condition subroutine `errsns` is *not* provided, because it is totally specific to the VMS operating system. The terminate program subroutine `exit` was already provided by the operating system.

A sample implementation of time functions that may appear on other systems:

```
      subroutine startclock
      common / myclock / mytime
      integer mytime
      integer time
      mytime = time()
      return
      end
      function wallclock
      integer wallclock
      common / myclock / mytime
      integer mytime
      integer time
      integer newtime
      newtime = time()
      wallclock = newtime - mytime
      mytime = newtime
      return
      end
      integer wallclock, elapsed
      character*24 greeting
      real dtime
      real timediff, timearray(2)
c     print a heading
      call fdate( greeting )
      write( 6, 10 ) greeting
10    format('1hi, it''s ', a24 /)
c     see how long an ‘ls' takes, in seconds
      call startclock
      call system( 'ls' )
      elapsed = wallclock()
      write( 6, 20 ) elapsed
20    format(//,'elapsed time ', i4, ' seconds'///)
c     now test the cpu time for some trivial computing
      timediff = dtime( timearray )
      q = 0.01
      do 30 i = 1, 1000
          q = atan( q )
30    continue
      timediff = dtime( timearray )
      write( 6, 40 ) timediff
40    format(//,'computing atan(q) 1000 times',
&         /   'took ', f6.3,' seconds.'/)
      end
```

9.3 Formats

Some f77 format features may be different from the formats provided in other versions of FORTRAN 77. Even when the formats used in other FORTRAN 77 implementations are different, with a little care, programs are still often transportable to f77.

Here are some format specifiers that f77 treats differently than some other implementations:

- A—Used with character type data elements. In FORTRAN 77, this specifier worked with any variable type. f77 supports the older usage, up to four characters to a word.
- $—Suppress newline character output.
- R—Set an arbitrary radix for the I formats that follow in the descriptor.
- SU—Select unsigned output for following I formats. For example, you can convert output to either hexadecimal or octal with the following formats, instead of using the Z or O edit descriptors:

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

9.4 Carriage-Control

FORTRAN 77 carriage-control grew out of the capabilities of the equipment used when FORTRAN 77 was originally developed. For similar historical reasons, an operating system, derived from the UNIX operating system, does not have FORTRAN 77 carriage-control, but you can simulate it in two ways.

- For simple jobs, use OPEN(N, FORM='PRINT'). You then get single or double spacing, formfeed, and stripping off of column one. It is legal to reopen unit 6 to change the form parameter to PRINT, for example:

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

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

- Use the `asa` filter to transform FORTRAN 77 carriage-control conventions into the UNIX carriage-control format (see the `asa` (1) man page) before printing files with the `lpr` command.

9.5 File Equates

Early versions of FORTRAN 77 did not use named files, and file equates provided some ability to open files by name. You can use pipes and I/O redirection, as well as hard or soft links, in place of file equates in transported programs.

Example: This example uses `csh`(1). Redirect `stdin` from `redir.data`:

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

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

demo% f77 redir.f  ← The compile
redir.f:
 MAIN:
demo% a.out < redir.data ← Run with redirection
  9 9.90000
demo%
```

See Chapter 3, “File System and FORTRAN 77 I/O” for more on piping and redirection.

9.6 Data Representation

Read the appendix, “Data Representations,” in the *FORTRAN 77 4.0 Reference Manual* for the exact representation of different kinds of data in FORTRAN 77. This section points out information necessary for transporting FORTRAN 77 programs. Remember the following caveats:

- Because we adhere to the IEEE 754 standard for floating-point, the first four bytes in a `REAL*8` are not the same as in a `REAL*4`.

- The default sizes for reals, integers, and logicals are the same according to the FORTRAN 77 Standard, except when the `-i2` flag is used, which shrinks integers and logicals to two bytes, but leaves reals as four bytes.
- Character variables can be freely mixed and equivalenced with variables of other types, but be careful of potential alignment problems.
- SPARC system floating-point arithmetic does raise exceptions on overflow or divide-by-zero, but does not signal `SIGFPE` by default. It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. See the appendix, "Data Representations," in the *FORTRAN 77 4.0 Reference Manual*.
- The extreme finite, normalized values can be determined. See `libm_single`(3f) and `libm_double`(3f). The indeterminate forms can be written and read, using formatted and list-directed I/O statements.

9.7 Hollerith

This section is useful for porting older programs, not for writing or heavily modifying a program. It is recommended that you use character variables for this purpose. You can initialize variables with the older FORTRAN 77 Hollerith (*n*H) feature, but this is not standard practice.

Table 9-3 Maximum Characters in Data Types

	Maximum Number of Standard ASCII Characters				
Data Type	**No** `-i2`**,** `-i4`**,** `-r8`**,** `-dbl`	`-i2`	`-i4`	`-r8`	`-dbl`
`BYTE`	1	1	1	1	1
`COMPLEX`	8	8	8	16	16
`COMPLEX*16`	16	16	16	16	16
`COMPLEX*32`	32	32	32	32	32
`DOUBLE COMPLEX`	16	16	16	32	32
`DOUBLE PRECISION`	8	8	8	16	16
`INTEGER`	4	2	4	4	8
`INTEGER*2`	2	2	2	2	2
`INTEGER*4`	4	4	4	4	4

Table 9-3 Maximum Characters in Data Types *(Continued)*

Data Type	**Maximum Number of Standard ASCII Characters** No -i2, -i4, -r8, -dbl	-i2	-i4	-r8	-dbl
INTEGER*8	*needs* -dbl	8	8	8	8
LOGICAL	4	2	4	4	8
LOGICAL*1	1	1	1	1	1
LOGICAL*8	*needs* -dbl	8	8	8	8
REAL	4	4	4	8	8
REAL*4	4	4	4	4	4
REAL*8	8	8	8	8	8
REAL*16	16	16	16	16	16

For storing standard ASCII characters with normal Fortran:

- With -r8, unspecified size INTEGER and LOGICAL do *not* hold double.
- With -dbl, unspecified size INTEGER and LOGICAL *do* hold double.

That is, the storage is there with both options, but is unavailable in normal Fortran with -r8.

Example: Initialize variables with Hollerith:

```
      double complex x(2)
      data x /16HHello there, sai, 16Hlor, new in town/
      write( 6, '(4A8, "?")' ) x
      end
```

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

If you must, you can initialize a data item of a compatible type with a Hollerith, and then pass it around.

Example:

```
      integer function doyouloveme()
      double precision fortran, beloved
      integer yes, no
      data yes, no / 3hyes, 2hno /
      data fortran/ 7hFORTRAN/
10    format( "Whom do you love? ", $ )
      write( 6, 10 )
      read ( 5, 20 ) beloved
20    format( a8 )
      doyouloveme = no
      if ( beloved .eq. fortran ) doyouloveme = yes
      return
      end
```

```
      program trouble
      integer yes, no
      integer doyouloveme
      data yes, no / 3hyes, 2hno /

      if ( doyouloveme() .eq. yes ) then
          print *, 'You are sick'
      else
          print *, 'See if I ever speak to you again'
      endif
      end
```

All these constructs produce warning messages from the compiler.

9.8 Porting Steps

The following outline of steps leads into performance issues, which is the topic of the next chapter, but does not contain *all* that you need to know about porting. It is designed for someone who must do a large job in a short time, and who does not code in FORTRAN 77 regularly.

Typical Case

Here is a sample situation:

- The code is of modest size (10K lines).
- All the subroutines are contained in one file.
- A simple command line: `f77 -O prog.f,` does not work.

What to do?

1. **For your own protection, first save a complete set of the original files, including any `README` and `.COM` files.**

2. **Make a new directory, say `src`, and copy your files to it, and go there.**

```
demo% mkdir src
demo% cd src
```

3. **Split the one file with many subroutines into many files, one subroutine per file.**

```
demo% fsplit ../prog.f
```

This command may produce a lot of files. `fsplit` may not always work, so do not delete `prog.f`.

4. Create a makefile.

```
FFLAGS = -fast $(FLAGS)
OBJ = subs.o main.o

example: $(OBJ)
    f77 $(FFLAGS) $(OBJ) -pg -o example \
        -Bstatic -lm
```

Performance issues start about here.

If the `-pg` option is placed on the `ld` line, it results in a profile that does not include the columns, `#calls time/call`, because the individual routines are not compiled with `-pg`. That is why `-pg` is on the compile line.

5. Compile all the source files with one makefile command.

```
demo% make
```

Since we selected `–fast` as the default compilation flag in the makefile, we have implicitly asked for the `–O3` level of optimization, among other options.

6. Execute the code.

```
demo% example
```

7. Check the answers; make sure they are correct.

8. Run `gprof`.

```
demo% gprof example > profile
```

Examine the profile reports of `gprof`, using `more` or your editor of choice.

The report comes in two parts, a flat profile and a call graph report. The flat report comes second, and can be found by searching for the "flat" string.

You may want to recompile the most expensive routines (those coming first in the flat report) with `–fast –O4`. Compile either by hand, or by editing the makefile. A simplistic makefile rewrite looks like this:

```
FFLAGS = -fast $(FLAGS)
OBJ = subs.o main.o

example: $(OBJ)
    f77 $(FFLAGS) $(OBJ) -pg -o example \
    -Bstatic -lm

expensive_routine.o: expensive_routine.f
    f77 -fast -O4 -c expensive_routine
```

If the answers are correct and the timing information is fast enough, that is. within about 20% of your target, you have completed the job. If it is not fast enough, tune the code.

Troubleshooting

Here are a few troubleshooting tips.

If the Answers Are Close, but Not Right On

Do the following:

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

  VAX math is not as good as IEEE math, and even different IEEE processors may differ. This is especially true if it involves many trig functions. These functions are much more complicated than one might think, and the standard defines only the basic arithmetic functions, so there can be subtle differences, even between IEEE machines.

- Try running with `call nonstandard_arithmetic`. Doing so can also improve performance considerably, and make your Sun workstation behave more like a VAX. If you have a VAX or some other computer handy, run it there, also. It is quite common for many numerical applications to produce slightly different results on each floating-point implementation.
- Check for `NaN`, `+Inf`, and other signs of probable errors. See “IEEE Routines” or the `man` page `ieee_handler`(3m) for instructions on how to trap the various exceptions. On most machines, these exceptions simply abort the run.
- Two numbers can differ by 6 x 10^{29} but have the same floating-point form. Here is an example of different numbers, but the same representation:

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

The output is:

```
99,999,990 x 10^29 = 0.99999993E+37 = 7cf0bdc1
99,999,996 x 10^29 = 0.99999993E+37 = 7cf0bdc1
```

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

If the Program Fails without Warning

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

- Turn off the optimizer. If the program then works, turn the optimizer back on for only the top routines.
- Understand that optimizers must make assumptions about the program. If you have done some nonstandard things, like using the SAVE statement, they can cause problems. Almost no optimizer handles *all* programs at *all* levels of optimization.

Before calling for help, make sure you have the current software, such as FORTRAN 77 4.0 and Solaris 2.x or Solaris 1.x, and you are either under warranty or have a software support contract.

Profiling 10

This chapter is organized into the following sections:

This chapter describes how to measure the resources used by programs.

10.1 Examples

This following program is used in several examples. It is a revised version of the one in Chapter 7, “Debugging,” and it calls `mkidentity` 100,000 times.

Example: Function for profiling:

p3.f

```
      real function determinant(m)
      real m(2,2)
      determinant = m(1,1) * m(2,2) - m(1,2) * m(2,1)
      return
      end
```

Example: Main for profiling:

p1.f

```
      program silly
      paramater (n=2)
      real twobytwo(2,2) / 4 *-1 /
      do i = 1, 100000
           call mkidentity( twobytwo, n )
      end do
      print *, determinant(twobytwo)
      end
```

Example: Subroutine for profiling:

p2.f

```
      subroutine mkidentity(matrix,dim)
      real matrix(dim,dim)
      integer dim
      do 90 m = 1, dim
           do 20 n = 1, dim
                if(m.eq.n) then
                        matrix(m,n) = 1.
                else
                        matrix(m,n) = 0.
                endif
20         continue
90    continue
      return
      end
```

10.2 The `time` Command

The simplest way to gather data about the resources consumed by a program is to use the `time` (1) command, or, in `csh`, the `set time` command.

Example

Let's compile the above sample program with or without `-g`, and run `time` on it. The output format may vary.

```
demo% f77 -o silly -silent p1.f p2.f p3.f
Linking:
demo% time silly
     1.00000
3.2u 0.3s 0:08 41% 0+104k 0+0io 0pf+0w
demo%
```

The interpretation is:

- 3.2 seconds on user code
- 0.3 seconds executing system code on behalf of the user
- 0 minutes and 8 seconds to complete
- 41% of the machine's resources dedicated to this program, approximately
- 0 kilobytes of program memory, 104 kilobytes of data memory (averages)
- 0 reads and 0 writes
- 0 page faults
- 0 swapouts

If there is I/O, the output is similar to this:

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
```

The interpretation is:

- 6 seconds on user code, approximately
- 17 seconds on system code on behalf of the user, approximately
- 1 minute 16 seconds to complete
- 31% of the resources dedicated to this program
- 11 kilobytes of shared program memory
- 21 kilobytes of private data memory

- 354 reads
- 210 writes
- 135 page faults
- 0 swapouts

iMPact FORTRAN 77 MP Notes

If iMPact FORTRAN 77 MP is used, the number from `/bin/time` is interpreted in a different way. Since `/bin/time` accumulates the user time on different threads, the user number is no longer used, and only real time is used.

Since the user time displayed includes the time spent on all the processors, it can be quite large, and is not a good measure of performance. A better measure is the real time, which is the wall clock time.

Also, since the real time is the wall clock time, if you run the parallel version of the benchmark, avoid running too many programs at the same time.

10.3 The `gprof` *Command*

The `gprof` (1) command provides a detailed procedure-by-procedure analysis of execution time, including how many times a procedure was called, who called it and who it called, and how much time was spent in the procedure and by the routines that it called.

Compiling and Linking

First, compile and link the program with the `-pg` flag:

```
demo% f77 -o silly -silent -pg p1.f p2.f p3.f
Linking:
demo%
```

Execution

To obtain meaningful timing information, execution must complete normally.

```
demo% silly
    1.00000
demo%
```

After execution completes, a file named `gmon.out` is written in the working directory. This file contains profiling data that can be interpreted with `gprof`.

The `gprof` Utility

Run the `gprof` utility on the program, `silly`. `gprof` produces about 14 pages of report for this short program.

The report is mostly two profiles of how the total time is distributed across the program procedures: the call graph and the flat profile. They are preceded by an explanation of the column labels, followed by an index.

In the following graph profile, the line that begins with `[4]` is called the *function line*; the lines above it, the *parent lines*; and the lines below it, the *descendant* lines.

Only the first few lines of some sections are shown in the following table.

```
demo% gprof silly
     @(#)callg.blurb 1.5 88/02/08 SMI
 call graph profile:
    …
                                   called/total       parents
 index  %time    self descendents  called+self    name    index
                                   called/total       children

                                                      <spontaneous>
 [1]     99.5    0.00        3.82                 start [1]
                 0.00        3.82       1/1           _main [3]
                 0.00        0.00       1/1           _finitfp_ [303]
                 0.00        0.00       1/1           _on_exit [314]
 -----------------------------------------------
                 0.15        3.67       1/1           _main [3]
 [2]     99.5    0.15        3.67       1         _MAIN_ [2]
                 3.67        0.00  100000/100000      _mkidentity_ [4]
                 0.00        0.00       1/1           _s_wsle [317]
                 0.00        0.00       1/1           _determinant_ [296]
                 0.00        0.00       1/1           _do_l_out [297]
                 0.00        0.00       1/1           _e_wsle [299]
 -----------------------------------------------
                 0.00        3.82       1/1           start [1]
 [3]     99.5    0.00        3.82       1         _main [3]
                 0.15        3.67       1/1           _MAIN_ [2]
                 0.00        0.00      16/16          _signal [254]
                 0.00        0.00       1/1           _f_init [302]
                 0.00        0.00       1/1           __enable_sigfpe_master [277]
                 0.00        0.00       1/1           _ieee_retrospective_ [308]
                 0.00        0.00       1/1           _f_exit [301]
                 0.00        0.00       1/1           _exit [300]
 -----------------------------------------------
                 3.67        0.00  100000/100000      _MAIN_ [2]
 [4]     95.6    3.67        0.00  100000         _mkidentity_ [4]
 -----------------------------------------------
    …
demo%
```

Function Line

The function line in the example above reveals that:

- mkidentity was called 100,000 times.
- 3.67 seconds were spent in mkidentity itself.
- 0 second was spent in routines called by mkidentity.
- 95.6% of the execution time of silly is from mkidentity.

Parent line

The single parent line reveals that MAIN was the only procedure to call mkidentity, that is, all 100,000 invocations of mkidentity came from MAIN. Thus, all of the 3.67 seconds spent in mkidentity were spent on behalf of MAIN.

If mkidentity had also been called from another procedure, there would be two parent lines, and the 3.67 seconds of *self* time would be divided between MAIN and the other caller. The descendant lines are interpreted similarly.

Overhead

When you enable profiling, the running time of a program may significantly increase. The fact that mcount, the utility routine used to gather the raw profiling data, is usually at the top of the flat profile shows this.

To eliminate this overhead in the completed version of the program, recompile all the source files without the -pg option. Ignore the overhead incurred by mcount when interpreting the flat profile. The graph profile attempts to automatically subtract time attributed to mcount when computing percentages of total runtime. The result may not be accurate due to UNIX timekeeping conventions.

The FORTRAN 77 library includes two routines that return the total time used by the calling process. See dtime(3F) and etime(3F).

10.4 The tcov Command

The tcov (1) command provides a detailed statement-by-statement profile of an actual test case of a program.

Compiling and Linking

First, compile and link it with -a, as in this example. This example uses -a on all modules, but it is usually better to use this option on only those modules which profiling has shown to be most expensive.

```
demo% f77 -silent -o silly -a p1.f p2.f p3.f
```

Execution

To generate meaningful timing information, execution must complete normally, or the user code must call exit(2).

```
demo% silly
1.00000
demo%
```

After execution completes, there is a new file named p1.tcov in the working directory. This file contains profiling data that can be interpreted with tcov.

The tcov Utility

Run the tcov utility on the source file, p1.f:

```
demo% tcov p1.f
```

Then list `p1.tcov`:

```
demo% cat p1.tcov
           program silly
           parameter (n=2)
           real twobytwo(2,2) / 4 *-1 /
    1 ->    do i = 1, 100000
    100000 -> call mkidentity( twobytwo, n )
          end do
    1 ->   print *, determinant(twobytwo)
           end
        Top 10 Blocks

    Line    Count
    5       100000
    4            1
    7            1
    3      Basic blocks in this file
    3      Basic blocks executed
    100.00 Percent of the file executed
    100002 Total basic block executions
    33334.00Average executions per basic block
demo%
```

10.5 I/O Profiling

You can obtain a report about how much data was transferred by your program. For each FORTRAN 77 unit, the report shows the file name, the number of I/O statements, the number of bytes, and some statistics on these items.

To obtain a I/O profiling report:

1. **Insert the statement,** `external start_iostats`**, before the first executable statement, and insert a call to** `start_iostats` **before the first I/O statement that you want to measure.**

```
    external start_iostats
    …
    call start_iostats
```

I/O statements profiled include READ, WRITE, PRINT, OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, and REWIND. The runtime system opens stdin, stdout, and stderr before the first executable statement of your program, so you must reopen these units after the call to start_iostats, without first closing them.

Example: Profile stdin, stdout, and stderr:

```
      EXTERNAL start_iostats
      …
      CALL start_iostats
      OPEN(5)
      OPEN(6)
      OPEN(0)
```

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

2. **Compile with the -pg option and run your program.**

```
demo% f77 -pg src.f
demo% a.out
```

3. View the report file.

If the executable file name is *name*, the report is on the *name*`.io_stats` file.

```
demo% cat a.out.io_stats
            Input Report
1. unit    2. file name            3. input data          4. map
                           cnt     total    avg   std dev    (cnt)
---------------------------------------------------------------------
      0         stderr      0         0    0.0      0.00    No
                            0         0    0.0      0.00
      5          stdin      0         0    0.0      0.00    No
                            0         0    0.0      0.00
      6         stdout      0         0    0.0      0.00    No
                            0         0    0.0      0.00
     10           temp      0         0    0.0      0.00    No
     …
            Output Report
1. unit           5. output data         6. blk size  7. fmt  8. direct
           cnt     total    avg   std dev                       (rec len)
--------------------------------------------------------------------------
      0      0         0    0.0      0.00           0     Yes     seq
             0         0    0.0      0.00
      5      0         0    0.0      0.00           0     Yes     seq
             0         0    0.0      0.00
      6      1         3    3.0      0.00           0     Yes     seq
             1         3    3.0      0.00
     10   2000      8000    4.0      0.00       16384     Yes     dir
     …
```

10.6 Missing Profile Libraries

If the profiling libraries are not installed, and if you try to use profiling, you may get an error message like this:

```
demo% f77 -p real.f
real.f:
 MAIN stuff:
ld: -lc_p: No such file or directory
demo%
```

There is a system utility to extract files from the release CD. You can use it to get the debugging files after the system is installed. See `add_services`(8). You may want to get help from your system administrator.

Performance 11

This chapter introduces performance and optimizing issues. Most of the references that are cited delve into the subject far more deeply than this chapter. This chapter is organized into the following sections.

For a helpful mind set, remember that:

- There can be no cookbook for tuning.
- There is no substitute for experience and human cleverness. Many tactics can and must be employed.
- The best I/O is no I/O.
- You should concentrate on the big picture. Solve the real problem.
- A cycle here and a cycle there *in a key loop* add up to many mips.
- Code tuning is not for the squeamish nor the faint of heart.
- It can be exciting—but frustrating.

11

11.1 Why Tune Code?

There are two situations where code tuning is important:

- Benchmarking
- Application porting

11.2 Algorithm Choice

Algorithm choice is critical, and is *always* made on the basis of machine architecture.

In olden times (1950-1970), all machines were scalar. Most were 32-60 bits, with extended precision accumulators. Memory was expensive. Therefore, old algorithms were inner-product based, that is, dot-product based, like the Crout reduction, Cholesky Decomposition, and so forth. `sqrt` was expensive, but improved numerical properties of the algorithms that were employed, so it allowed more problems to be run in single precision.

With the advent of the CRAY-1, vector algorithms became the rage. Dot products were replaced with SAXPY operations. New constraints on algorithms came about due to the difference between what computer scientists and mathematicians thought constituted a vector operation. In general, a vector algorithm does more work than a similar scalar algorithm.

Actually, SAXPY became popular somewhat before the advent of vector machines. Dot product formulations tend to march through memory in the natural way (through the rows of each column) for one matrix, and the other way (through the columns of each row) for the other. On some high-performance scalar machines of that era, this change resulted in suboptimal performance due to cache affects. SAXPY allows each matrix to be addressed in the natural fashion, at the cost of doing somewhat more memory accesses and losing some accuracy, due to the failure to accumulate in extended-precision registers. On vector machines, SAXPY is always the preferred technique because vector performance is really devastated by dot product formulations.

For example: Best incore sort: 300 < n < 700:
scalar ⟶ quicksort
vector ⟶ Pangali's bubble sort

The Pangali bubble sort does a lot more work, but executes 15 to 20 times faster on many vector machines.

Since most of us do not have vector machines, why worry about vector algorithms? One reason is that the user code may include attempts at complex vectorized algorithms. If you can replace complex vectorized code with simple scalar code, you can do less work and run faster. It can be much easier to concentrate on the underlying science and worry less about the programming.

11.3 Tuning Methodology

Get the program to run and generate correct answers. Do not apply any tricks until you have correct results.

If the run is long, say longer than 20 minutes, and it is obvious how to reduce the problem size, do it. Rerun and save the output to be used as correct. If the code runs for a very long time (many hours), you must do this. If it runs for 22 minutes and changing the problem is not easy, skip to the next step.

Examine the profile. Recompile the top routines (say 80% of the total time) with the `-a` switch to enable `tcov` analysis. Also toss in the `-pg` switch to count the number of calls and time spent in the routine. Don't throw away the optimized (good) versions. You may have uses for them.

Rerun. It will take a little longer. Do not bother to obtain a *dry* machine, as this will not matter.

Run `gprof` and `tcov`. See the `man` pages or Chapter 10, "Profiling."

Compare the `gprof` with the regular run. Have the routines maintained their relative order? If so, continue without reservation. If not, work on routines in the order of their original import.

Consider the subroutine, `COSTSaLOT`:

```
      subroutine COSTSaLOT(randvec,n)
      real randvec(n)
      do i = 1, n
         randvec(i) = random() ! user random no. generator
      end do
      return
      end
```

`tcov` shows the following:

```
        subroutine COSTSaLOT(randvec,n)
        real randvec(n)

     1 ->  do i = 1, n
1000000 ->  randvec(i) = random()
          end do
     1 ->  return
          end

        real function random()

1000000 -> random = rand(0)
        return
        end
```

The trick here is to *inline* the random number generator; that is, rewrite the program as:

```
    subroutine COSTSaLOT(randvec,n)
    real randvec(n)
    do i = 1, n
       randvec(i) = rand(0)
    end do
    return
    end
```

On a vector machine, it is generally better to inline the code of `rand` itself, and then this is close to optimal. On SPARC systems, it may be better *not* to compute a whole vector at a time. Since there is limited cache, it may be better to remove `COSTSaLOT` entirely, and simply call `rand(0)` from the calling program.

We've learned from this exercise that:

- `tcov` is handy for pinpointing exactly where to work.
- We should try to inline small subroutines to reduce call overhead.
- Thinking that the best solution is to precompute a lot of things, does not make it so.

11.4 Loop Jamming

Start with a double loop like the following:

```
do i = 1, n
    stuff
end do
do i = 1, n
    more stuff
end do
```

You can rewrite the loop like this:

```
do i = 1, n
    stuff
    more stuff
end do
```

This loop can be a fair win on SPARC systems. It can also be a loss, depending on cache sizes, SCRAM[1], and other considerations.

So, time it and try it. Always take the loop in question and run it in isolation. Experimentation works. Bu*t concentrate* on the loops that consume the most time. It is often necessary to run some profiler program on the code.

This list only scratches the surface. Once you have narrowed down the expensive sections, it is easy to ask for assistance.

Do not forget to think about the algorithm—are you computing the best way?

1. SCRAM is on only the Sun-4 / 110, and stands for Static Column Random Access Memory.

11.5 Benchmark Case History

Consider the following trigonometric function benchmark:

test.f

```
      program test
      integer*4 limit, i, n
      parameter (limit=100000)
      double precision hold(3,limit), x1, x2, x3
      do 10 i = 1, limit
          do 5 n = 1, 3
              hold(n,i) = 0.0
5         continue
10    continue
      x1 = 0.0
      x2 = 0.0
      x3 = 0.0
      open( 3, file='test.tmp', form='FORMATTED' )
      do 20 i = 1, limit
          x1 = x1 + 1
          hold(1,i) = x1
          x2 = sin(hold(1,i)) - cos(hold(1,i))
          hold(2,i) = x2
          x3 = sqrt(hold(1,i)**2 + hold(2,i)**2)
          hold(3,i) = x3
          write(3,*) (hold(n,i),n = 1, 3)
          if (  x3 .le. 0.00001d0 .and.
              x3 .ge. -0.00001d0 ) then
             write(3,*) 'x3 = 0.0'
          elseif ( x2 .le. 0.00001d0 .and.
                 x2 .ge. -0.00001d0 ) then
             write(2,*) 'x2 = 0.0'
          else
              x2 = atan(x2/x3)
              x1 = hold(2,i) * hold(3,i)
          endif
20    continue
      close(3)
      end
```

This is one minute too slow, compared to some particular computer, so recompile with the `-p` profiling option, and then profile the code with the `prof` utility.

```
demo% f77 -p -O3 test.f
test.f:
 AIN test:
demo% prof a.out
```

The output from `prof` is:

```
time a.out
real    4m19.36s
user    4m1.00s
sys     0m4.05s
prof
 %time  cumsecs  #call  ms/call  name
  24.0    58.32                  mcount
  10.2    83.10 499995     0.05  __fp_rightshift
   7.8   102.15 100000     0.19  _s_wsle
   7.3   119.96                 .div
   7.2   137.42     64   272.81.rem
   4.7   148.874600144     0.00.umul
   4.0   158.65 300000     0.03  _unpacked_to_decimal
   3.9   168.09 300000     0.03  _wrt_F
   3.8   177.24 300000     0.03  __fp_leftshift
   3.0   184.44 300000     0.02  _fconvert
   2.6   190.881499992     0.00  __fourdigits
   2.6   197.28 300000     0.02  _binary_to_decimal_fraction
   2.4   203.081199996     0.00  __mul_10000
   1.6   207.01 299996     0.01  _binary_to_decimal_integer
   1.6   210.886299964     0.00.urem
   1.3   213.95                  _sincos
   1.1   216.57                  _MAIN_
   1.1   219.18 299996     0.01  __fp_normalize
   1.1   221.77                  _nwrt_A
     … many more lines …
   0.0   243.24      1     0.00  _strcpy
   0.0   243.24      1     0.00  _strlen
   0.0   243.24      3     0.00  _t_runc
demo%
```

You can also use `-pg` and `gprof`. Examples are shown in Chapter 10, “Profiling.”

What can you tell from this profile?

`mcount` is taking much of the CPU. Therefore, the program spends more time jumping between modules than computing. The user code is very simple, however. Optimization was high (O3/4), and in-lining was turned on. So where is the time going?

Notice that the top routines are `.mul`, `.div`, `.rightshift`, and the user code is not doing that. Furthermore, `sincos`, which is usually one of the most used routines, accounts for only 2% of the runtime.

From this analysis, you can infer that something else, aside from trigonometric calculations, is a performance issue. The output file shows:

```
-rw-r--r-- 1 khb 5800000 Jan 25 13:02 test.tmp
```

The file is quite large.

Now examine the code. Note that *every* time through the loop, it writes to the output file. If the program were large, you may have to recompile with `-a`, and run `tcov` to catch this phenomenon.

Modify the code. Eliminate not only the write, but the *unnecessary* `if` tests by commenting them out, for example.

```
cccccccc write(3,*) (hold(n,i),n = 1, 3)
cccccccc if ( x3 .le. 0.00001d0 .and.
cccccccc   x3 .ge. -0.00001d0 ) then
cccccccc    write(3,*) 'x3 = 0.0'
cccccccc elseif ( x2 .le. 0.00001d0 .and.
cccccccc     x2 .ge. -0.00001d0 ) then
cccccccc    write(2,*) 'x2 = 0.0'
         x2 = atan(x2/x3)
         x1 = jold(2,i) * hold(3,i)
cccccccc endif
```

Why unnecessary? Because on any IEEE machine, such as a SPARC system, there is no difficulty in computing x/0.0. It is ±`Inf` or 0.0/0.0 (`NaN`), or a bad `atan`. In real applications, you can do a large chain of operations, and only need to check the final result by using `libm_single` and `libm_double` routines for IEEE handling, and perhaps `ieee_flags` to condition the exception flags. Doing so can remove *millions* of `if` tests; that is, one `if` test in a key loop is executed *millions* of times.

This is the output from `prof`:

```
real    0m9.65s           profiled times
user    0m4.35s
sys     0m0.60s

 %time  cumsecs  #call  ms/call  name
  62.3     2.71                  _sincos {much more sensible!}
  36.6     4.30                  _MAIN_
   0.5     4.32                  _cos
   0.5     4.34                  _sin
   0.2     4.35      5     2.00  _ioctl
   0.0     4.35     64     0.00 .rem
   0.0     4.35      1     0.00 .udiv
   0.0     4.35      3     0.00 .umul
   0.0     4.35      1     0.00  __enable_sigfpe_master
   0.0     4.35      1     0.00  __findiop
   0.0     4.35      1     0.00  _access
   0.0     4.35      2     0.00  _bzero
   0.0     4.35      2     0.00  _calloc
   0.0     4.35      4     0.00  _canseek
   0.0     4.35      4     0.00  _close
   0.0     4.35      1     0.00  _exit

     ... many more lines ...

   0.0     4.35      1     0.00  _strcpy
   0.0     4.35      1     0.00  _strlen
   0.0     4.35      2     0.00  _t_runc
demo%
```

Run it again without profiling, for optimal reporting time:

```
real 0m5.20s
user 0m4.28s
sys  0m0.61s
```

From this example, we have learned that:

- Performance analysis and tuning are iterative processes. Think about what you can do differently:
  - Use different compile options.
  - Profile, but profile carefully; don't jump to conclusions.
  - If necessary, use `prof` and `tcov` to find out what is really happening.
- If you can get a 20-times speedup by changing the code, do it. The IEEE arithmetic is new enough that not everyone knows how to use it to good advantage. Knowing why it is good and what it is good for can really help. It can be the start of a commitment to state-of-the-art standards.

11.6 Optimization

At optimization level `–O4`, the compiler inlines calls to functions and subroutines which are defined in the same file as the caller. Thus, the usual UNIX advice of splitting each function and subroutine into a separate file may adversely impact performance. It may require experimentation with collecting different modules in different files to achieve maximum performance.

11.7 References

The following reference books provide more details:

- *FORTRAN 77 4.0 Reference Manual*, SunSoft, Inc.
- *Numerical Computation Guide*, SunSoft, Inc.
- *Performance Tuning an Application*, SunSoft, Inc.
- *Programming Pearls,* by Jon Louis Bentley, Addison Wesley
- *More Programming Pearls,* by Jon Louis Bentley, Addison Wesley
- *Writing Efficient Programs,* by Jon Louis Bentley, Prentice Hall
- *FORTRAN Optimization,* by Michael Metcalf, Academic Press 1982
- *Optimizing FORTRAN Programs,* by C. F. Schofield Ellis Horwood Ltd., 1989
- *A Guidebook to Fortran on Supercomputers,* Levesque, Williamson, Academic Press, 1989

C–FORTRAN 77 Interface 12

This chapter is organized into the following sections:

Glendower: I can call spirits from the vasty deep.
Hotspur: Why, so can I, or so can any man;
But will they come when you do call for them?
Henry IV, Part I

12.1 Sample Interface

As an introductory example, a FORTRAN 77 main calls a C function:

Samp.c

```
samp ( i, f )
    int *i;
    float *f;
{
    *i = 9;
    *f = 9.9;
}
```

In the above program, both `i` and `f` are pointers.

Sampmain.f

```
      integer i
      real r
      external Samp !$pragma C ( Samp )
      call Samp ( i, r )
      write( *, "(I2, F4.1)") i, r
      end
```

Both `i` and `f` are passed by reference, which is the default.

Compile and execute, with output:

```
demo% cc -c Samp.c
demo% f77 -silent Samp.o Sampmain.f
demo% a.out
 9 9.9
demo%
```

12.2 How to Use this Chapter

We suggest you use this chapter in the following manner:

1. **Examine the above example and the section, “Getting It Right.”**
2. **Read the section, “FORTRAN 77 Calls C,” or “C Calls FORTRAN 77.”**
3. **Within that section, choose one of these subsections:**

- Arguments passed by reference
- Arguments passed by value
- Function return values
- Labeled common
- Sharing I/O
- Alternate returns

4. **Within that subsection, choose one of these examples:**

   For the arguments, there is an example for each of these, or a note that it cannot be done.

- Simple types (`character*1`, `logical`, `integer`, `real`, `double precision`, `quad`)
- Complex types (`complex`, `double complex`)
- Character strings (`character*n`)
- One-dimensional arrays (`integer a(9)`)
- Two-dimensional arrays (`integer a(4,4)`)
- Structured records (structure and record)
- Pointers

  For *function return values*, there is an example for each of these:
  - Integer (`int`)
  - Real (`float`)
  - Pointer to real (pointer to `float`)
  - Double precision (`double`)
  - Quadruple precision (`long double`)
  - Complex
  - Character string

  For each of *labeled common*, *sharing I/O*, and *alternate returns*, there is one set of examples. These are the same for "FORTRAN 77 calls C" or" C calls FORTRAN 77."

12.3 Getting It Right

Most C/FORTRAN 77 interfaces must be correct in all of these aspects:

- Function/subroutine: definition and call
- Data types: compatibility of types
- Arguments: passing by reference or value
- Arguments: order
- Procedure name: uppercase and lowercase and trailing underscore (_)
- Libraries: telling the linker to use FORTRAN 77 libraries

Some C/FORTRAN 77 interfaces must also be correct on these constructs:

- Arrays: indexing and order
- File descriptors and `stdio`
- File permissions

Function or Subroutine

The word *function* have different meanings in C and FORTRAN 77:

- In C, all subprograms are functions; it is just that some of them return a null value.
- In FORTRAN 77, a function passes a return value, but a subroutine does not.

FORTRAN 77 Calls a C Function

If the called C function returns a value, call it from FORTRAN 77 as a function.

If the called C function does not return a value, call it as a subroutine.

C Calls a FORTRAN 77 Subprogram

If the called FORTRAN 77 subprogram is a *function*, call it from C as a function that returns a comparable data type.

If the called FORTRAN 77 subprogram is a *subroutine*, call it from C as a function that returns a value of `int` (comparable to FORTRAN 77 `INTEGER*4`) or `void`. This return value is useful if the FORTRAN 77 routine does a nonstandard return.

Data Type Compatibility

Data types have the following sizes and alignments without –f, –i2, -misalign, –r4, or –r8.

Table 12-1 Argument Sizes and Alignments—Pass by Reference

FORTRAN 77 Type	C Type	Size (Bytes)	Alignment (Bytes)
BYTE X	char x	1	1
CHARACTER X	char x	1	1
CHARACTER**n* X	char x[n]	n	1
COMPLEX X	struct {float r,i;} x;	8	4
COMPLEX*8 X	struct {float r,i;} x;	8	4
DOUBLE COMPLEX X	struct {double dr,di;}x;	16	4
COMPLEX*16 X	struct {double dr,di;}x;	16	4
COMPLEX*32 X *(SPARC only)*	struct {long double dr,di;} x;	32	4
DOUBLE PRECISION X	double x	8	4
REAL X	float x	4	4
REAL*4 X	float x	4	4
REAL*8 X	double x	8	4
REAL*16 X *(SPARC only)*	long double x	16	4
INTEGER X	int x	4	4
INTEGER*2 X	short x	2	2
INTEGER*4 X	int x	4	4
INTEGER*8 X	long long int x	8	4
LOGICAL X	int x	4	4
LOGICAL*1 X	char x	1	1
LOGICAL*2 X	short x	2	2
LOGICAL*4 X	int x	4	4
LOGICAL*8 X	long long int x	8	8

Note the following:

- The REAL*16 and the COMPLEX*32 can be passed between f77 and ANSI C, but not between f77 and some previous versions of C.
- Alignments are for f77 types.
- Arrays pass by reference, if the elements are compatible.
- Structures pass by reference, if the fields are compatible.

- Passing arguments by value:
  - You cannot pass arrays, character strings, or structures by value.
  - You can pass arguments by value from `f77` to C, but not from C to `f77`, since the `%VAL()` does not work in a `SUBROUTINE` statement.

Case Sensitivity

C and FORTRAN 77 take opposite perspectives on case sensitivity:

- C is case sensitive—uppercase or lowercase matters.
- FORTRAN 77 ignores case.

The `f77` default is to ignore case by converting subprogram names to lowercase. It converts all uppercase letters to lowercase letters, except within character-string constants.

There are two usual solutions to the uppercase/lowercase problem:

- In the C subprogram, make the name of the C function all lowercase.
- Compile the `f77` program with the `-U` option, which tells `f77` to preserve existing uppercase/lowercase distinctions, that is, not to convert to all lowercase letters.

Use one or the other, but not both.

Most examples in this chapter use all lowercase letters for the name in the C function, and do *not* use the `f77 –U` compiler option.

Underscore in Names of Routines

The FORTRAN 77 compiler normally appends an underscore (_) to the names of subprograms for both a subprogram and a call to a subprogram. This convention distinguishes it from C procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All FORTRAN 77 library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are three usual solutions to the underscore problem:

- In the C function, change the name of the function by appending an underscore to that name.

- Use the `C()` pragma to tell the FORTRAN 77 compiler to omit those trailing underscores.
- Use the `-ext_names` option to make external names without underscores.

  See “-ext_names=e” on page 47, for more information.

Use one of these solutions, but not all three.

Most of the examples in this chapter use the FORTRAN 77 `C()` compiler pragma, and do *not* use the underscores. The `C()` pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C language, so the FORTRAN 77 compiler does not append an underscore to such names, as it ordinarily does with external names. The `C()`directive for a particular function must appear before the first reference to that function. It must also appear in each subprogram that contains such a reference. The conventional usage is:

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

If you use this pragma, then in the C function, you must *not* append an underscore to those names.

Argument-Passing by Reference or Value

In general, FORTRAN 77 passes arguments by reference. In a call, if you enclose an argument with the nonstandard function `%VAL()`, FORTRAN 77 passes it by value.

In general, C passes arguments by value. If you precede an argument by an ampersand (&), C passes it by reference. C always passes arrays and character strings by reference.

Arguments and Order

For every argument of character type, an argument is passed giving the length of the value. The string lengths are equivalent to C `long int` quantities, passed by value.

The order of arguments is:

- Address for each argument (datum or function)
- A `long int` for each character argument. The whole list of string lengths comes after the whole list of other arguments.

Example: A FORTRAN 77 call in a code fragment:

```
CHARACTER*7 S
INTEGER B(3)
 …
CALL SAM( B(2), S )
```

The above call is equivalent to the C call in this code fragment:

```
char s[7];
long b[3];
  …
sam_( &b[1], s, 7L ) ;
```

Array Indexing and Order

Array indexing and order work in the following manner.

Array Indexing

C arrays always start at zero, but by default, FORTRAN 77 arrays start at 1. There are two usual ways of approaching indexing.

- You can use the FORTRAN 77 default, as in the above example. Then the FORTRAN 77 element `B(2)` is equivalent to the C element `b[1]`.
- You can specify that the FORTRAN 77 array `B` starts at 0. as follows:

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

This way, the FORTRAN 77 element `B(1)` is equivalent to the C element `b[1]`.

Array Order

FORTRAN 77 arrays are stored in column-major order, C arrays in row-major order. For one-dimensional arrays, this is no problem. For two-dimensional and higher arrays, switch subscripts in all references and declarations.

Tip

Some may find it confusing to, say, triangularize in C and then pass the parts to FORTRAN 77. More generally, it may be confusing to do some of the matrix manipulation in C and some in FORTRAN 77.

So, if passing parts of arrays between C and FORTRAN 77 does not work, or if it is confusing, try passing the *whole* array to the other language and do *all* the matrix manipulation there. Avoid doing part in C and part in FORTRAN 77.

Libraries and Linking with the `f77` Command

To link the proper FORTRAN 77 libraries, use the `f77` command to pass the `.o` files on to the linker. Doing so usually shows up as a problem only if a C main calls FORTRAN 77. *Dynamic* linking is encouraged and made easy.

Example 1: Use `f77` to link:

```
demo% f77 -c -silent RetCmplx.f
demo% cc -c RetCmplxmain.c
demo% f77 RetCmplx.o RetCmplxmain.o ← This command line does the linking.
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

Example 2: Use `cc` to link. A failure occurs; the libraries are not linked.

```
demo% f77 -silent -c RetCmplx.f
demo% cc RetCmplx.o RetCmplxmain.c  ←  Wrong link command
ld: Undefined symbol  ←  missing routine
 __Fc_mult
demo%
```

File Descriptors and `stdio`

FORTRAN 77 I/O channels are in terms of unit numbers. The I/O system does not deal with unit numbers, but with *file descriptors*. The FORTRAN 77 runtime system translates from one to the other, so most FORTRAN 77 programs do not have to recognize file descriptors.

Many C programs use a set of subroutines, called *standard I/O* (or `stdio`). Many functions of FORTRAN 77 I/O use standard I/O, which in turn uses operating system I/O calls. Some of the characteristics of these I/O systems are listed in the following table.

Table 12-2 Characteristics of Three I/O Systems

	FORTRAN 77 Units	**Standard I/O File Pointers**	**File Descriptors**
Files Open	Opened for reading and writing	Opened for reading; or Opened for writing; or Opened for both; or Opened for appending. See `OPEN`(3S).	Opened for reading; or Opened for writing; or Opened for both
Attributes	Formatted or unformatted	Always unformatted, but can be read or written with format-interpreting routines	Always unformatted
Access	Direct or sequential	Direct access if the physical file representation is direct access, but can always be read sequentially	Direct access if the physical file representation is direct access, but can always be read sequentially
Structure	Record	Character stream	Character stream
Form	Arbitrary nonnegative integers	Pointers to structures in the user's address space	Integers from 0-63

File Permissions

C programmers traditionally open input files for reading and output files for writing, sometimes for both. In FORTRAN 77, it is not possible for the system to foresee what use you will make of the file, since there is no parameter to the `OPEN` statement that gives that information.

FORTRAN 77 tries to open a file with the maximum permissions possible, first for both reading and writing, then for each separately.

This event occurs transparently and is of concern only if you try to perform a `READ`, `WRITE`, or `ENDFILE`, but you do not have permission. Magnetic tape operations are an exception to this general freedom, since you can have write permissions on a file, but not have a write ring on the tape.

12.4 FORTRAN 77 Calls C

This section covers arguments passed by reference or value, functions, common blocks, sharing I/O, and alternate returns.

Arguments Passed by Reference (`f77` Calls C)

This subsection covers simple types, complex types, strings, and arrays.

Simple Types Passed by Reference (`f77` Calls C)

For simple types, define each C argument as a pointer:

SimRef.c

```
simref ( t, f, c, i, r, d, q, si )
   char    * t, * f, * c ;
   int     * i ;
   float   * r ;
   double  * d ;
   long double  * q ;
   short   * si ;
{
        *t = 1 ; *f = 0 ;
        *c  = 'z' ;
        *i  = 9 ;
        *r  = 9.9 ;
        *d  = 9.9 ;
        *q  = 9.9 ;
        *si = 9 ;
}
```

Default: Pass each FORTRAN 77 argument by reference:

SimRefmain.f

real*16 is SPARC only

```
        logical*1  t, f
        character  c
        integer    i*4, si*2
        real       r*4, d*8, q*16
        external SimRef !$pragma C( SimRef )
        call SimRef ( t, f, c, i, r, d, q, si )
        write(*, "(L2,L2,A2,I2,F4.1,F4.1,F4.1,I2)")
&                  t, f, c, i, r,   d,   q,   si
        end
```

Compile and execute, with output:

```
demo% cc -c SimRef.c
demo% f77 -silent SimRef.o SimRefmain.f
demo% a.out
 T F z 9 9.9 9.9 9.9 9
demo%
```

Complex Types Passed by Reference (`f77` Calls C)

Here, the C argument is a pointer to a structure:

CmplxRef.c

```
cmplxref ( w, z )
     struct complex { float r, i; } *w;
     struct dcomplex { double r, i; } *z;
{
     w -> r = 6;
     w -> i = 7;
     z -> r = 8;
     z -> i = 9;
}
```

CmplxRefmain.f

```
      complex w
      double complex z
      external CmplxRef !$pragma C( CmplxRef )
      call CmplxRef ( w, z )
      write(*,*) w
      write(*,*) z
      end
```

Compile and execute, with output:

```
demo% cc -c CmplxRef.c
demo% f77 -silent CmplxRef.o CmplxRefmain.f
demo% a.out
  ( 6.00000, 7.00000)
  ( 8.0000000000000, 9.0000000000000)
demo%
```

Character Strings Passed by Reference (`f77` Calls C)

Passing strings between C and FORTRAN 77 is not encouraged.

The rules are:

- All C strings pass by reference.
- For each FORTRAN 77 argument of character type, an *extra argument* is passed giving the length of the string. The extra argument is equivalent to a C `long int` passed by value. This rule is nonstandard.
- The order of arguments is:

1. A list of the regular arguments
2. A list of lengths, one for each character argument, each as a `long int`
3. The list of extra arguments comes after the list of regular arguments.

Example: Character strings passed by reference. A FORTRAN 77 call:

```
    CHARACTER*7 S
    INTEGER B(3)
…
    CALL SAM( B(2), S )
```

The above call is equivalent to the C call in

```
    char s[7];
    long b[3];
…
    sam_( &b[1], s, 7L );
```

Ignoring the Extra Arguments of Passed Strings

You can ignore the extra arguments, since they are after the list of other arguments. The following C function ignores the extra arguments:

StrRef.c

```
strref ( s10, s80 )
    char *s10, *s80;
{
    static char ax[11] = "abcdefghij";
    static char sx[81] = "abcdefghijklmnopqrstuvwxyz";
    strncpy ( s10, ax, 11 );
    strncpy ( s80, sx, 26 );
}
```

The following FORTRAN 77 call generates hidden extra arguments:

StrRefmain.f

```
    character s10*10, s80*80
    external StrRef !$pragma C( StrRef )
    call StrRef( s10, s80 )
    write (*, 1) s10, s80
1   format("s10='", A, "'", / "s80='", A, "'")
    end
```

Compile and execute, with output:

```
demo% cc -c StrRef.c
demo% f77 -silent StrRef.o StrRefmain.f
demo% a.out
s10='abcdefghij'
s80='abcdefghijklmnopqrstuvwxyz'
demo%
```

Using the Extra Arguments of Passed Strings

You can *use* the extra arguments.

The following C function *uses* the extra arguments. It prints the lengths.

StrRef2.c

```
strref ( s10, s80, L10, L80 )
    char *s10, *s80 ;
    long L10, L80 ;
{
    static char ax[11] = "abcdefghij" ;
    static char sx[81] = "abcdefghijklmnopqrstuvwxyz" ;
    printf("%d %d \n", L10, L80 ) ;
    strncpy ( s10, ax, 11 ) ;
    strncpy ( s80, sx, 26 ) ;
}
```

If you compile `StrRef2.c` and `StrRefmain.f`, then you get this output:

```
10 80
s10='abcdefghij'
s80='abcdefghijklmnopqrstuvwxyz'
```

One-Dimensional Arrays Passed by Reference (f77 Calls C)

A C array, indexed from 0 to 8:

FixVec.c

```
fixvec ( V, Sum )
    int *Sum;
    int V[9];
{
    int i;
    *Sum = 0;
    for ( i = 0; i <= 8; i++ ) *Sum = *Sum + V[i];
}
```

A FORTRAN 77 array, implicitly indexed from 1 to 9:

FixVecmain.f

```
    integer i, Sum
    integer a(9) / 1,2,3,4,5,6,7,8,9 /
    external FixVec !$pragma C( FixVec )
    call FixVec ( a, Sum )
    write(*, '(9I2, " ->" I3)') (a(i),i=1,9), Sum
    end
```

Compile and execute, with output:

```
demo% cc -c FixVec.c
demo% f77 -silent FixVec.o FixVecmain.f
demo% a.out
 1 2 3 4 5 6 7 8 9 -> 45
demo%
```

A FORTRAN 77 array, explicitly indexed from 0 to 8:

FixVecmain2.f

```
    integer i, Sum
    integer a(0:8) / 1,2,3,4,5,6,7,8,9 /
    external FixVec !$pragma C( FixVec )
    call FixVec ( a, Sum )
    write(*, '(9I2, " ->" I3)') (a(i),i=0,8), Sum
    end
```

Compile and execute, with output:

```
demo% cc -c FixVec.c
demo% f77 -silent FixVec.o FixVecmain2.f
demo% a.out
 1 2 3 4 5 6 7 8 9 -> 45
demo%
```

Two-Dimensional Arrays Passed by Reference (`f77` Calls C)

In a two-dimensional array, the rows and columns are switched.

Example: A 2-by -2 C array, indexed from 0 to 1 and 0 to 1:

FixMat.c

```
fixmat ( a )
     int a[2][2];
{
     a[0][1] = 99;
}
```

A 2-by-2 FORTRAN 77 array, explicitly indexed from 0 to 1, and 0 to 1:

FixMatmain.f

```
   integer c, m(0:1,0:1) / 00, 10, 01, 11 /, r
   external FixMat !$pragma C ( FixMat )
   do r = 0, 1
       do c = 0, 1
          write(*,'("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
       end do
   end do
   call FixMat ( m )
   do r = 0, 1
       do c = 0, 1
          write(*,'("m(",I1,",",I1,")=",I2.2)') r, c, m(r,c)
       end do
   end do
   end
```

Compile and execute. Show m before and after the C call:

Compare a[0][1] with m(1,0):
C changed a[0][1], which is FORTRAN 77 m(1,0).

```
demo% cc -c FixMat.c
demo% f77 -silent FixMat.o FixMatmain.f
demo% a.out
m(0,0) = 00
m(0,1) = 01
m(1,0) = 10
m(1,1) = 11
m(0,0) = 00
m(0,1) = 01
m(1,0) = 99
m(1,1) = 11
demo%
```

Structured Records Passed by Reference (f77 Calls C)

Example: A C structure of an integer and a character string:

StruRef.c

```
struct VarLenStr {
    int nbytes;
    char a[26];
};
void
struchr ( v )
struct VarLenStr *v;
{
    bcopy( "oyvay", v->a, 5 );
    v->nbytes = 5;
}
```

A FORTRAN 77 structured record of an integer and a character string:

StruRefmain.f

```
      structure /VarLenStr/
          integer nbytes
          character a*25
      end structure
      record /VarLenStr/ vls
      character s25*25
      external StruChr !$pragma C( StruChr )
      vls.nbytes = 0
      Call StruChr( vls )
      s25(1:5) = vls.a(1:vls.nbytes)
      write ( *, 1 ) vls.nbytes, s25
1     format("size =", I2, ", s25='", A, "'" )
      end
```

Compile and execute, with output:

```
demo% cc -c StruRef.c
demo% f77 -silent StruRef.o StruRefmain.f
demo% a.out
size = 5, s25='oyvay'
demo%
```

Pointers Passed by Reference (`f77` Calls C)

To C, it is a pointer to a pointer:

`PassPtr.c`

```
passptr ( i, d )
    int **i;
    double **d;
{
    **i = 9;
    **d = 9.9;
}
```

FORTRAN 77 passes by reference, and it is passing a pointer:

`PassPtrmain.f`

```
    program PassPtrmain
    integer          i
    double precision d
    pointer ( iPtr, i ), ( dPtr, d )
    external PassPtr !$pragma C( PassPtr )
    iPtr = malloc( 4 )
    dPtr = malloc( 8 )
    i = 0
    d = 0.0
    call PassPtr ( iPtr, dPtr )
    write( *, "(i2, f4.1)" )  i, d
    end
```

Compile and execute, with output:

```
demo% cc -c PassPtr.c
demo% f77 -silent PassPtr.o PassPtrmain.f
demo% a.out
 9 9.9
demo%
```

Arguments Passed by Value (`f77` Calls C)

In the call, enclose an argument in the nonstandard function `%VAL()`. This rule works for all simple types and pointers.

Simple Types Passed by Value (`f77` Calls C)

If you prototype the float parameter, C does not promote to double.

`SimVal.c`

```
simval ( char t, char c, int i, float r, double d,
    long double q, short s, int *reply )
{
        *reply = 0 ;
        /* If nth arg ok, set nth octal digit to one */
        if ( t         ) *reply = *reply + 1 ;
        if ( c == 'z'  ) *reply = *reply + 8 ;
        if ( i == 9    ) *reply = *reply + 64 ;
        if ( r == 9.9F ) *reply = *reply + 512 ;
        if ( d == 9.9  ) *reply = *reply + 4096 ;
        if ( q == 9.9L ) *reply = *reply + 32768 ;
        if ( s == 9    ) *reply = *reply + 262144 ;
}
```

Pass each FORTRAN 77 argument by value, except for `args`:

`SimValmain.f`

REAL*16 is SPARC only

```
        logical*1  t
        character  c
        integer    i*4, s*2, args*4
        real       r*4, d*8, q*16
        data t / .true. /,  c / 'z' /
&            i/ 9 /,  r/9.9/, d/ 9.9D0 /,  q/ 9.9Q0 /,  s/ 9 /
        external SimVal !$pragma C( SimVal )
        call SimVal( %VAL(t), %VAL(c), %VAL(i),
&              %VAL(r), %VAL(d), %VAL(q), %VAL(s), args )
        write( *, 1 )  args
 1      format('args=', o7, '(If nth digit=1, arg n OK)')
        end
```

Compile and execute, with output:

```
demo% cc -c SimVal.c
demo% f77 -silent SimVal.o SimValmain.f
demo% a.out
args=1111111(If nth digit=1, arg n OK)
demo%
```

Complex Types Passed by Value (`f77` Calls C)

You can pass the `complex` structure by value:

CmplxVal.c

```
cmplxval ( w, z )
    struct complex { float r, i; } w, *z;
{
    z->r = w.r * 2.0;
    z->i = w.i * 2.0;
    w.r = 0.0;
    w.i = 0.0;
}
```

CmplxValmain.f

```
      complex w / ( 4.0, 4.5 ) /
      complex z
      external CmplxVal !$pragma C( CmplxVal )
      call CmplxVal ( %VAL(w), z )
      write(*,*) w
      write(*,*) z
      end
```

Compile and execute, with output:

```
demo% cc -c CmplxVal.c
demo% f77 -silent CmplxVal.o CmplxValmain.f
demo% a.out
  ( 4.00000, 4.50000)
  ( 8.00000, 9.00000)
demo%
```

Arrays, Strings, Structures Passed by Value (`f77` Calls C) - N/A

You cannot pass arrays, character strings, or structures by value—at least there is no reliable way that works on all architectures. The workaround is to pass them by reference.

Pointers Passed by Value (`f77` Calls C)

C receives the argument as a pointer.

PassPtrVal.c

```
passptrval ( i, d  )
    int     *i ;
    double   *d ;
{
    *i  = 9 ;
    *d = 9.9 ;
}
```

FORTRAN 77 passes a pointer by value:

PassPtrValmain.f

```
    program PassPtrValmain
    integer          i
    double precision d
    pointer ( iPtr, i ), ( dPtr, d )
    external PassPtrVal !$pragma C( PassPtrVal )
    iPtr = malloc( 4 )
    dPtr = malloc( 8 )
    i = 0
    d = 0.0
    call PassPtrVal ( %VAL(iPtr), %VAL(dPtr) ) ! Nonstandard
    write( *, "(i2, f4.1)" )  i, d
    end
```

Compile and execute, with output:

```
demo% cc -c PassPtrVal.c
demo% f77 -silent PassPtrVal.o PassPtrValmain.f
demo% a.out
 9 9.9
demo%
```

Function Return Values (f77 Calls C)

For function return values, a FORTRAN 77 function of type BYTE, INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or REAL*16 (quadruple precision) is equivalent to a C function that returns the corresponding type. There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

Return an int (f77 Calls C)

RetInt.c

```
int retint ( r )
int *r;
{
    int s;
    s = *r;
    s++;
    return ( s );
}
```

RetIntmain.f

```
    integer r, s, RetInt
    external RetInt !$pragma C( RetInt )
    r = 8
    s = RetInt ( r )
    write( *, "(2I4)") r, s
    end
```

Compile, link, and execute, with output:

```
demo% cc -c RetInt.c
demo% f77 -silent RetInt.o RetIntmain.f
demo% a.out
 8 9
demo%
```

In the same way, do a function of type BYTE, LOGICAL, REAL, or DOUBLE PRECISION. Use matching types according to Table 12-1.

Return a `float` *(*`f77` *Calls C)*

Return a float as follows:

RetFloat.c

```
float  retfloat ( pf )
float *pf ;
{
    float  f ;
    f = *pf ;
    f++ ;
    return ( f ) ;
}
```

RetFloatmain.f

```
    real  RetFloat, R, S
    external RetFloat !$pragma C( RetFloat )
    R = 8.0
    S = RetFloat ( R )
    print *, R, S
    end
```

```
demo% cc -c RetFloat.c
demo% f77 -silent RetFloat.o RetFloatmain.f
demo% a.out
    8.00000 9.00000
demo%
```

In earlier versions of C, if C returned a function value that was a `float`, C promoted it to a `double`, and various workarounds were necessary.

Return a Pointer to a `float` *(*`f77` *Calls C)*

This example shows how to return a function value that is a pointer to a `float`. Compare it with the previous example.

RetPtrF.c

```
static float f;
float *retptrf ( a )
float *a;
{
    f = *a;
    f++;
    return &f;
}
```

RetPtrFmain.f

```
    integer RetPtrF
    external RetPtrF !$pragma C( RetPtrF )
    pointer ( P, S )
    real R, S
    R = 8.0
    P = RetPtrF ( R )
    print *, S
    end
```

Compile and execute, with output:

```
demo% cc -c RetPtrF.c
demo% f77 -silent RetPtrF.o RetPtrFmain.f
demo% a.out
9.00000
demo%
```

Since the function return value is an address, you can assign it to the pointer value, or possibly do some pointer arithmetic. You *cannot* use it in an expression with, say, reals, such as `RetPtrF(R)+100.0`.

Return a `DOUBLE PRECISION` *(*`f77` *Calls C)*

Here is an example of C returning a type `double` function value to a FORTRAN 77 `DOUBLE PRECISION` variable:

RetDbl.c

```
double retdbl ( r )
double *r;
{
    double s;
    s = *r;
    s++;
    return ( s );
}
```

RetDblmain.f

```
      double precision r, s, RetDbl
      external RetDbl !$pragma C( RetDbl )
      r = 8.0
      s = RetDbl ( r )
      write( *, "(2F6.1)") r, s
      end
```

Compile and execute, with output:

```
demo% cc -c RetDbl.c
demo% f77 -silent RetDbl.o RetDblmain.f
demo% a.out
   8.0 9.0
demo%
```

Return a Quadruple Precision (`f77` Calls C)

Example: C returns a `long double` to a FORTRAN 77 `REAL*16`.

`RetQuad.c` *(SPARC only)*

```
long double  retquad ( pq )
long double *pq ;
{
    long double  q ;
    q = *pq ;
    q++ ;
    return ( q ) ;
}
```

`RetQuadmain.f` *(SPARC only)*

```
    real*16  RetQuad, R, S
    external RetQuad !$pragma C( RetQuad )
    R = 8.0
    S = RetQuad ( R )
    write(*,'(2F6.1)') R, S
    end
```

Compile and execute, with output:

```
demo% cc -c RetQuad.c
demo% f77 -silent RetQuad.o RetQuadmain.f
demo% a.out
   8.0   9.0
demo%
```

Return a COMPLEX *(*f77 *Calls C)*

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine with an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN 77 function is:

```
    COMPLEX FUNCTION F (…)
```

The pattern for a corresponding C function is

```
f_ (temp, … )
struct { float r, i; } *temp;
```

Example: C returns a type COMPLEX function value to FORTRAN 77:

RetCmplx.c

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

RetCmplxmain.f

```
    complex u, v, RetCmplx
    external RetCmplx !$pragma C( RetCmplx )
    u = ( 7.0, 8.0 )
    v = RetCmplx ( u )
    write( *, * ) u
    write( *, * ) v
    end
```

Compile and execute, with output:

```
demo% cc -c -silent RetCmplx.c
demo% f77 -silent RetCmplx.o RetCmplxmain.f
demo% a.out
  ( 7.00000, 8.00000)
  ( 8.00000, 9.00000)
demo%
```

Return a Character String (`f77` Calls C)

Passing strings between C and FORTRAN 77 is not encouraged. A character-string-valued FORTRAN 77 function is equivalent to a C function with the two extra initial arguments—data address and length.

A FORTRAN 77 function of this form, with no `C()` pragma is:

```
    CHARACTER*15 FUNCTION G ( … )
```

The above FORTRAN 77 function is equivalent to a C function of this form:

```
g_ ( result, length, … )
char result[ ];
long length;
```

In either form, the function can be invoked in C with this call:

```
char chars[15];
…
    g_ ( chars, 15L, … );
```

Example: No pragma:

RetStr.c

```
retstr_ ( retval_ptr, retval_len, ch_ptr, n_ptr, ch_len )
char *retval_ptr, *ch_ptr;
int retval_len, *n_ptr, ch_len;
{
    int count, i;
    char *cp;
    count = *n_ptr;
    cp = retval_ptr;
    for (i=0; i<count; i++) {
        *cp++ = *ch_ptr;
    }
}
```

In the above example:

- The returned string is passed by the extra arguments, `retval_ptr` and `retval_len`, a pointer to the start and length of the string.
- The character-string argument is passed with `ch_ptr` and `ch_len`.
- The `ch_len` is at the end of the argument list.
- The repeat factor is passed as `n_ptr`.

In FORTRAN 77, use the above C function from `RetStr.c`, as follows:

RetStrmain.f

```
    CHARACTER String*100, RetStr*50
    String = RetStr ( '*', 10 )
    PRINT *, "'", String(1:10), "'"
    END
```

The output from `RetStrmain.f` is:

```
demo% cc -c RetStr.c
demo% f77 -silent RetStr.o RetStrmain.f
demo% a.out
'**********'
demo%
```

Labeled Common (`f77` Calls C)

C and FORTRAN 77 can share values in labeled common.

The C function:

`UseCom.c`

The method is the same no matter which language calls which.

```
extern struct comtype { /* Define a  structure  appropriate for this common */
    float p ;
    float q ;
    float r ;
    } ;
extern struct comtype ilk_ ; /* Establish the labeled common  */

void
usecom ( int *count )         /* Like the SUBROUTINE statement  */
{
    *count = 3 ;
    ilk_.p = 7.0 ;
    ilk_.q = 8.0 ;
    ilk_.r = 9.0 ;
}
```

FORTRAN 77 main program (labeled common):

`UseCommain.f`

```
    INTEGER n
    REAL u, v, w
    COMMON / ilk / u, v, w
    EXTERNAL UseCom !$pragma C( UseCom )
    n = 3
    u = 1.0
    v = 2.0
    w = 3.0
    CALL UseCom ( n )
    WRITE(*,"(' u =', F4.1, ', v =', F4.1, ', w =', F4.1)") u,v,w
    END
```

Compile and execute, with output:

Any of the options that change size or alignment (or any equivalences that change alignment) may invalidate such sharing.

```
demo% f77 -c -silent UseCommain.f
demo% cc -c UseCom.c
demo% f77 UseCom.o UseCommain.o
demo% a.out
 u = 7.0, v = 8.0, w = 9.0
demo%
```

Sharing I/O (`f77` Calls C)

Mixing FORTRAN 77 I/O with C I/O is not recommended. If you must mix them, it is usually safer to pick one and stick with it, rather than alternating.

The FORTRAN 77 I/O library is implemented largely on top of the C standard I/O library. Every open unit in a FORTRAN 77 program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

If a FORTRAN 77 main program calls C, then before the FORTRAN 77 program starts, the FORTRAN 77 I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C function must take the FORTRAN 77 I/O environment into consideration to perform I/O on open file descriptors.

Mixing with `stdout` (`f77` Calls C)

A C function that writes to `stderr` and to `stdout` is shown as follows:

`MixIO.c`

```
#include <stdio.h>
mixio ( n )
int *n;
{
    if ( *n <= 0 ) {
        fprintf ( stderr, "error: negative line #\n" );
        *n = 1;
    }
    printf ( "In C: line # = %d \n", *n );
}
```

In FORTRAN 77, use the above C function as follows:

MixIOmain.f

```
      integer n / -9 /
      external MixIO !$pragma C( MixIO )
      do i = 1, 6
          n = n + 1
          if ( abs(mod(n,2)) .eq. 1 ) then
              call MixIO ( n )
          else
              write(*,'("In FORTRAN 77: line # =",i2)') n
          end if
      end do
      end
```

Compile and execute, with output:

```
demo% cc -c MixIO.c
demo% f77 -silent MixIO.o MixIOmain.f
demo% a.out
In FORTRAN 77: line # =-8
error: negative line #
In C: line # = 1
In FORTRAN 77: line # = 2
In C: line # = 3
In FORTRAN 77: line # = 4
In C: line # = 5
demo%
```

Mixing with `stdin` *(*`f77` *Calls C)*

A C function that reads from `stdin` is shown as follows:

MixStdin.c

```
#include <stdio.h>
int c_read_ ( fd, buf, nbytes, buf_len )
FILE **fd;
char *buf;
int *nbytes, buf_len;
{
     return fread ( buf, 1, *nbytes, *fd );
}
```

In FORTRAN 77, use the above C function, as follows:

MixStdinmain.f

```
      character*1 inbyte
      integer*4 c_read, getfilep
      external getfilep
      write(*,'(a,$)') 'What is the digit? '
      irtn = c_read ( getfilep(5), inbyte, 1 )
      write(*,9) inbyte
 9    format('The digit read by C is ', a )
      end
```

FORTRAN 77 does the prompt. C does the read:

```
demo% cc -c MixStdin.c
demo% f77 -silent MixStdin.o MixStdinmain.f
demo% a.out
What is the digit? 3
The digit read by C is 3
demo%
```

Alternate Returns (`f77` Calls C) - N/A

C does not have an alternate return. The workaround is to pass an argument and branch on that.

12.5 C Calls FORTRAN 77

This section covers arguments passed by reference or value, functions, common blocks, sharing I/O, and alternate returns.

Arguments Passed by Reference (C Calls `f77`)

This subsection covers simple types, complex types, strings, and arrays.

Simple Types Passed by Reference (C Calls `f77`)

FORTRAN 77 passes all these arguments by *reference* (default):

SimRef.f

REAL*16 is SPARC only

```
      subroutine SimRef ( t, c, i, si, r, d, q )
      logical*1  t
      character  c
      integer    i*4, si*2
      real       r*4, d*8, q*16
      t = .true.
      c  = 'z'
      i  = 9
      si = 9
      r  = 9.9
      d  = 9.9
      q = 9.9
      return
      end
```

C passes the address of each:

SimRefmain.c

```
main ( )
{
   char    t ;
   char    c ;
   int     i ;
   short   si ;
   float   r ;
   double  d ;
   long double  q = 5.5 ;
   extern simref_ ( char *t, char *c, int *i, short *si,
       float *r, double *d, long double *q ) ;
   simref_ ( &t, &c, &i, &si, &r, &d, &q ) ;
   printf ( "%08o %c %d  %d %3.1f %3.1f %L3.1f \n",
              t,   c, i, si,    r,    d,    q ) ;
}
```

Here are some simple types passed by reference:

```
demo% f77 -c -silent SimRef.f
demo% cc -c SimRefmain.c
demo% f77 SimRef.o SimRefmain.o    ← This command line does the linking.
demo% a.out
00000001 z 9  9 9.9 9.9 9.9
demo%
```

Complex Types Passed by Reference (C Calls f77)

The complex types require a simple structure:

CmplxRef.f

```
      subroutine CmplxRef ( w, z )
      complex w
      double complex z
      w = ( 6, 7 )
      z = ( 8, 9 )
      return
      end
```

In the above example, w and z are passed by reference (default).

CmplxRefmain.c

```
main ( )
{
    struct complex { float r, i; };
    struct complex d1;
    struct complex *w = &d1;
    struct dcomplex { double r, i; };
    struct dcomplex d2;
    struct dcomplex *z = &d2;
    extern cmplxref_ ();
    cmplxref_ ( w, z );
    printf ( "%3.1f %3.1f \n%3.1f %3.1f \n",
            w->r, w->i, z->r, z->i );
}
```

`w` and `z` are pointers, so if you pass `w` and `z`, you pass the address. This is passing by reference.

Compile and execute, with output:

```
demo% f77 -c -silent CmplxRef.f
demo% cc -c CmplxRefmain.c
demo% f77 CmplxRef.o CmplxRefmain.o
demo% a.out
6.0 7.0
8.0 9.0
demo%
```

Character Strings Passed by Reference (C Calls `f77`)

Passing strings between C and FORTRAN 77 is not encouraged.

Here are the rules for passing strings:

- All C strings pass by reference.
- For each FORTRAN 77 argument of character type, an *extra argument* is passed, giving the length of the string. The extra argument is equivalent to a C `long int` passed by value. This practice is nonstandard.
- The order of arguments is as follows:

1. A list of the regular arguments
2. A list of lengths, one for each character argument, as a `long int`
3. The list of extra arguments comes after the list of regular arguments

Example: Character strings passed by reference. A FORTRAN 77 call:

```
    CHARACTER*7 S
    INTEGER B(3)
…
    CALL SAM( B(2), S )
```

The above call is equivalent to the this C call:

```
    char s[7];
    long b[3];
…
    sam_( &b[1], s, 7L );
```

If you make a string in FORTRAN 77, you must provide an explicit null terminator because FORTRAN 77 does not automatically do that, and C expects it.

Ignoring the Extra Arguments of Passed Strings

You can *ignore* the extra arguments, since they are after the list of other arguments.

The following FORTRAN 77 subroutine gets no values of the extra arguments from the C main:

StrRef.f

```
    subroutine StrRef ( a, s )
    character a*10, s*80
    a = 'abcdefghi' // char(0)
    s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
    return
    end
```

The following C main ignores the extra arguments:

StrRefmain.c

```
main ( )
{
    char s10[10], s80[80];
    strref_ ( s10, s80 );
    printf ( " s10='%s' \n s80='%s' \n", s10, s80 );
}
```

In the above example, C strings pass by reference.

Compile and execute, with output:

```
demo% f77 -c -silent StrRef.f
demo% cc -c StrRefmain.c
demo% f77 StrRef.o StrRefmain.o
demo% a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
demo%
```

Using the Extra Arguments of Passed Strings

You can *use* the extra arguments.

The following FORTRAN 77 routine *uses* the extra arguments (the sizes) implicitly. The FORTRAN 77 source code cannot use them explicitly.

StrRef2.f

```
      subroutine StrRef2 ( a, s )
      character a*(*), s*(*)
      a = 'abcdefghi' // char(0)
      s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
      return
      end
```

The following C main passes the extra arguments explicitly:

StrRef2main.c

```
main ( )
{
    char s10[10], s80[80] ; /*Provide memory for the strings*/
    long  L10, L80 ;
    L10 = 10 ;              /*Initialize extra args*/
    L80 = 80 ;
    strref2_ ( s10, s80, L10, L80 ) ; /*pass extra args to f77*/
    printf ( " s10='%s' \n s80='%s' \n", s10, s80 ) ;
}
```

In the above example, C strings pass by reference.

Compile and execute, with output:

```
demo% f77 -c -silent StrRef2.f
demo% cc -c StrRef2main.c
demo% f77 StrRef2.o StrRef2main.o
demo% a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
demo%
```

Arguments Passed by Value (C Calls `f77`) - N/A

FORTRAN 77 can call C, and pass an argument by *value*. However, FORTRAN 77 cannot handle an argument passed by value if C calls FORTRAN 77. The workaround is to pass all arguments by *reference*.

Function Return Values (C Calls `f77`)

For function return values, a FORTRAN 77 function of type `BYTE`, `INTEGER`, `LOGICAL`, `DOUBLE PRECISION`, or `REAL*16` (quadruple precision) is equivalent to a C function that returns the corresponding type. There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

Return an `int` (C Calls `f77`)

Example: FORTRAN 77 returns an `INTEGER` function value to C:

`RetInt.f`

```
      integer function RetInt ( k )
      integer k
      RetInt = k + 1
      return
      end
```

RetIntmain.c

```
main()
{
    int k, m;
    extern int retint_ ();
    k = 8;
    m = retint_ ( &k );
    printf( "%d %d\n", k, m );
}
```

Compile and execute, with output:

```
demo% f77 -c -silent RetInt.f
demo% cc -c RetIntmain.c
demo% f77 RetInt.o RetIntmain.o
demo% a.out
8 9
demo%
```

Return a `float` *(C Calls* `f77`*)*

Example: FORTRAN 77 returns a `REAL` to a C `float`:

RetFloat.f

```
      real function RetReal ( x )
      real x
      RetReal = x + 1.0
      return
      end
```

RetFloatmain.c

```
main ( )
{
        float r, s ;
        extern  float retreal_ () ;
        r = 8.0 ;
        s = retreal_ ( &r ) ;
        printf( " %8.6f %8.6f \n", r, s ) ;
}
```

Compile and execute, with output:

```
demo% f77 -c -silent RetFloat.f
demo% cc -c RetFloatmain.c
demo% f77 RetFloat.o RetFloatmain.o
demo% a.out
 8.000000 9.000000
demo%
```

In earlier versions of C, if C returned a function value that was a `float`, C promoted it to a `double`, and various workarounds were necessary.

Return a `double` *(C Calls* `f77`*)*

Example: FORTRAN 77 returns a `DOUBLE PRECISION` function value to C:

RetDbl.f

```
      double precision function RetDbl ( x )
      double precision x
      RetDbl = x + 1.0
      return
      end
```

RetDblmain.c

```
main()
{
     double x, y;
     extern double retdbl_ ();
     x = 8.0;
     y = retdbl_ ( &x );
     printf( "%8.6f %8.6f\n", x, y );
}
```

Compile and execute, with output:

```
demo% f77 -c -silent RetDbl.f
demo% cc -c RetDblmain.c
demo% f77 RetDbl.o RetDblmain.o
demo% a.out
8.000000 9.000000
demo%
```

Return a `long double` *(C Calls* `f77`*)*

Example: FORTRAN 77 returns a REAL*16 to a C `long double`.

RetQuad.f

REAL*16 is SPARC only.

```
      real*16 function  RetQuad ( x )
      real*16 x
      RetQuad = x + 1.0
      return
      end
```

RetQuadmain.c

```
main ( )
{
        long double r, s ;
        extern  long double retquad_ ( long double * ) ;
        r = 8.0 ;
        s = retquad_ ( &r ) ;
        printf( " %8.6Lf %8.6Lf \n", r, s ) ;
}
```

Compile and execute, with output:

```
demo% f77 -c -silent RetQuad.f
demo% cc -c RetQuadmain.c
demo% f77 RetQuad.o RetQuadmain.o
demo% a.out
 8.000000 9.000000
demo%
```

Return a `COMPLEX` *(C Calls* `f77`*)*

A `COMPLEX` or `DOUBLE COMPLEX` function is equivalent to a C routine with an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN 77 function is shown here.

```
    COMPLEX FUNCTION F ( … )
```

The pattern for a corresponding C function is:

```
f_( temp, … )
struct { float r, i; } *temp;
```

Example: FORTRAN 77 returns a `COMPLEX` to a C `struct`:

RetCmplx.f

```
    complex function RetCmplx ( x )
    complex x
    RetCmplx = x * 2.0
    return
    end
```

RetCmplxmain.c

```
main ( )
{
    struct complex { float r, i; };
    struct complex c1, c2;
    struct complex *w = &c1, *t = &c2;
    extern retcmplx_ ();
    w -> r = 4.0;
    w -> i = 4.5;
    retcmplx_ ( t, w );
    printf ( " %3.1f %3.1f \n %3.1f %3.1f \n",
    w -> r, w -> i, t -> r, t -> i );
}
```

Return a `COMPLEX`. Compile, link, and execute, with output:

```
demo% f77 -c -silent RetCmplx.f
demo% cc -c RetCmplxmain.c
demo% f77 RetCmplx.o RetCmplxmain.o
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

Return a Character String (C Calls f77)

Passing strings between C and FORTRAN 77 is not recommended.

A FORTRAN 77 string function has two extra initial arguments—data address and length.

Example: A FORTRAN 77 function of the following form, with no `C()` pragma:

```
    CHARACTER*15 FUNCTION G ( … )
```

A C function of the following form:

```
g_ ( result, length, … )
char result[ ];
long length;
```

The above two functions are equivalent, and can be invoked in C as follows:

```
char chars[15];
g_ ( chars, 15L, … );
```

The lengths are passed by value. You must provide the null terminator.

`RetChr.f`

```
    FUNCTION RetChr( C, N )
    CHARACTER RetChr*(*), C
    RetChr = ''
    DO I = 1, N
       RetChr(I:I) = C
    END DO
    RetChr(N+1:N+1) = CHAR(0) ! Put in the null terminator.
    RETURN
    END
```

Return a character string (*Continued*):

RetChrmain.c

```
main()
{ /* Use a FORTRAN 77 character function, (C calls f77) */
        char strbuffer[9] = "123456789" ;
       char *rval_ptr = strbuffer ;      /* extra initial arg 1 */
      int rval_len = sizeof(strbuffer) ; /* extra initial arg 2 */
        extern void retchr_() ;
        char ch = '*' ;
        int n = 4 ;
        int ch_len = sizeof(ch) ;          /* extra final arg */
        printf( " '%s'\n", strbuffer ) ;
        retchr_ ( rval_ptr, rval_len, &ch, &n, ch_len ) ;
        printf( " '%s'\n", strbuffer ) ;
}
```

Compile, link, and execute, with output:

```
demo% f77 -c -silent RetChr.f
demo% cc -c RetChrmain.c
demo% f77 RetChr.o RetChrmain.o
demo% a.out
 '123456789'
 '****'
demo%
```

The caller must set up more actual arguments than are apparent as formal parameters to the FORTRAN 77 function:

- Arguments that are lengths of character strings are passed by *value*.
- Arguments that are *not* lengths of character strings are passed by *reference*.

Labeled Common (C Calls `f77`)

C and FORTRAN 77 can share values in labeled common**.** Here is a FORTRAN 77 subroutine:

`UseCom.f`

The method is the same, no matter which language calls which.

```
    SUBROUTINE UseCom ( n )
    INTEGER n
    REAL u, v, w
    COMMON / ilk / u, v, w
    n = 9
    u = 7.0
    v = 8.0
    w = 9.0
    RETURN
    END
```

The C main program:

`UseCommain.c`

```
#include <stdio.h>
extern struct comtype {  /* <-- Define a structure appropriate for this common. */
    float p ;
    float q ;
    float r ;
    } ;
extern struct comtype ilk_ ;   /* <-- Establish the labeled common. */
main()
{
    int count = 3 ;
    extern void usecom_ ( ) ;
    ilk_.p = 1.0 ;
    ilk_.q = 2.0 ;
    ilk_.r = 3.0 ;
    usecom_ ( &count ) ;   /* <--- This calls the subroutine. */
    printf(" ilk_.p=%4.1f, ilk_.q=%4.1f, ilk_.r=%4.1f\n",
    ilk_.p, ilk_.q, ilk_.r ) ;
}
```

Compile and execute, with output:

Any of the options that change the size or alignment (or any equivalences that change alignment) may invalidate such sharing.

```
demo% f77 -c -silent UseCom.f
demo% cc -c UseCommain.c
demo% f77 UseCom.o UseCommain.o
demo% a.out
 ilk_.p = 7.0, ilk_.q = 8.0, ilk_.r = 9.0
demo%
```

Sharing I/O (C Calls f77)

Mixing FORTRAN 77 I/O with C I/O is not recommended. If you must mix them, it is usually safer to pick one and stick with it, rather than alternating.

The FORTRAN 77 I/O library uses the C standard I/O library. Every open unit in a FORTRAN 77 program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

For sharing I/O, if a C main program calls a FORTRAN 77 subprogram, then there is no automatic initialization of the FORTRAN 77 I/O library that connects units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. If a FORTRAN 77 function attempts to reference the `stderr` stream (unit 0), then any output is written to a file named `fort.0`, instead of to the `stderr` stream.

To make the C program initialize I/O—establish the preconnection of units 0, 5, and 6—do the following:

1. **Insert the following line at the start of the C `main`:**

```
    f_init();
```

2. **At the end of the C `main`, insert the following line:**

```
    f_exit();
```

The second step may not be strictly necessary.

Example: Sharing I/O using a C main and a FORTRAN 77 subroutine:

MixIO.f

```
      subroutine MixIO ( n )
      integer n
      if ( n .LE. 0 ) then
          write(0,*) "error: negative line #"
          n = 1
      end if
      write(*,'("In FORTRAN 77: line # =",i2)') n
      end
```

MixIOmain.c

Insertion 1 ⟶

Insertion 2 ⟶

```
#include <stdio.h>
main ( )
{
    int i, m = -9;
    f_init();
        for ( i=0; i<=4; i++ ) {
        m++;
        if ( m == 2 || m == 4 )
            printf("In C: line # = %d\n",m);
        else
            mixio_ ( &m );
    }
    f_exit();
}
```

Compile and execute, with output:

```
demo% f77 -c -silent MixIO.f
demo% cc -c MixIOmain.c
demo% f77 MixIO.o MixIOmain.o
demo% a.out
error: negative line #
In FORTRAN 77: line # = 1
In C: line # = 2
In FORTRAN 77: line # = 3
In C: line # = 4
In FORTRAN 77: line # = 5
demo%
```

With a C main program, the following FORTRAN 77 library routines may not work correctly: `signal()`, `getarg()`, `iargc()`.

Alternate Returns (C Calls `f77`)

Some C programs need to use a FORTRAN 77 subroutine that has nonstandard returns. To C, such subroutines return an `int` (`INTEGER*4`). The return value specifies which alternate return to use. If the subroutine has no entry points with alternate return arguments, the returned value is undefined.

Example: One regular argument and two alternate returns:

`AltRet.f`

```
      subroutine AltRet ( i, *, * )
      integer i, k
      i = 9
      k = 20
      if ( k .eq. 10 ) return 1
      if ( k .eq. 20 ) return 2
      return
      end
```

C invokes the subroutine as a function:

`AltRetmain.c`

```
main()
{
    int k, m ;
    extern  int  altret_ () ;
    k = 0 ;
    m = altret_ ( &k ) ;
    printf( "%d %d\n", k, m ) ;
}
```

Compile, link, and execute:

```
demo% f77 -c -silent AltRet.f
demo% acc -c AltRetmain.c
demo% f77 AltRet.o AltRetmain.o
demo% a.out
9 2
demo%
```

In this example, the C `main` receives a 2 as the return value of the subroutine because `RETURN 2` has been executed.

Runtime Error Messages A

This appendix is organized into the following sections:

The f77 I/O library, signal handler, and operating system, when they are called by FORTRAN 77 routines, can all generate f77 error messages.

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 f77 library do not produce error messages directly. The following system routine in the f77 library calls C library routines which produce an error message:

```
      CALL SYSTEM("rm /")
      END
```

The following message is displayed:

```
rm: / directory
```

A.2 Signal Handler Error Messages

Before beginning execution of a program, the FORTRAN 77 library sets up a signal handler (`sigdie`) for signals that can cause termination of the program. `sigdie` prints a message that describes the signal, flushes any pending output, and generates a core image and a traceback.

Presently, the only arithmetic exception that produces an error message is the `INTEGER*2` division with a denominator of zero. All other arithmetic exceptions are ignored.

A signal handler error example follows, where the subroutine `SUB` tries to access parameters that are not passed to it:

```
      CALL SUB()
      END
      SUBROUTINE SUB(I,J,K)
      I=J+K
      RETURN
      END
```

The following error message results:

```
*** Segmentation violation
Illegal instruction (core dumped)
```

A.3 I/O Error Messages

The error messages in this section are generated by the FORTRAN 77 I/O library. The error numbers are returned in the `IOSTAT` variable if the `ERR` return is taken.

The following program tries to do an unformatted write to a file opened for formatted output:

```
WRITE( 6 ) 1
END
```

and produces error messages like the following:

```
sue: [1003] unformatted io not allowed
logical unit 6, named 'stdout'
lately: writing sequential unformatted external IO
```

The following error messages are generated. These same messages are also documented at the end of the `man` page, `perror`(3f).

If the error number is less than 1000, then it is a *system* error. See `intro` (2).

`1000 error in format`

Read the error message output for the location of the error in the format. It can be caused by more than 10 levels of nested parentheses or an extremely long format statement.

`1001 illegal unit number`

It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is 2^{31} - 1.

`1002 formatted io not allowed`

The logical unit was opened for unformatted I/O.

`1003 unformatted io not allowed`

The logical unit was opened for formatted I/O.

`1004 direct io not allowed`

The logical unit was opened for sequential access, or the logical record length was specified as 0.

`1005 sequential io not allowed`

The logical unit was opened for direct access I/O.

`1006 can’t backspace file`

You cannot do a `seek` on the file associated with the logical unit; therefore, you cannot backspace. The file may be a `tty` device or a pipe.

`1007 off beginning of record`

You tried to do a left tab to a position before the beginning of an internal input record.

`1008 can't stat file`

The system cannot return status information about the file. Perhaps the directory is unreadable.

`1009 no * after repeat count`

Repeat counts in list-directed I/O must be followed by an `*` with no blank spaces.

`1010 off end of record`

A formatted write tried to go beyond the logical end-of-record. An unformatted read or write also causes this.

`1011` <*Not used*>

`1012 incomprehensible list input`

List input has to be as specified in the declaration.

`1013 out of free space`

The library dynamically creates buffers for internal use. You ran out of memory for them; that is, your program is too big.

`1014 unit not connected`

The logical unit was not open.

`1015 read unexpected character`

Certain format conversions cannot tolerate nonnumeric data.

`1016 illegal logical input field`

`logical` data must be `T` or `F`.

`1017 'new' file exists`

You tried to open an existing file with `status='new'`.

`1018 can’t find 'old' file`

You tried to open a nonexistent file with `status='old'`.

`1019 unknown system error`

This error should not happen, but...

`1020 requires seek ability`

You tried to do a `seek` on a file that does not allow that. Some of the ways of performing an I/O operation that require a `seek` are:

- Direct access
- Sequential unformatted I/O
- Tabbing left

`1021 illegal argument`

Certain arguments to `open` and related functions are checked for legitimacy. Often only nondefault forms are checked.

`1022 negative repeat count`

The repeat count for list-directed input must be a positive integer.

`1023 illegal operation for unit`

You tried to do an I/O operation that is not possible for the device associated with the logical unit. You get this error if you try to read past end-of-tape, or end-of-file.

`1024` <*Not used*>

`1025 incompatible specifiers in open`

You tried to open a file with the `'new'` option and the `access='append'` option, or some other invalid combination.

`1026 illegal input for namelist`

A namelist read encountered an invalid data item.

`1027 error in FILEOPT parameter`

Using `OPEN`, the `FILEOPT` string has a bad syntax.

For example, the following error message is printed:

```
open: [1027] error in FILEOPT parameter
logical unit 8, named 'temp'
Abort
```

XView Toolkit B

This appendix is organized into the following sections:

This appendix introduces the `f77` interface to the XView programmer's toolkit, for FORTRAN 77 4.0 and OpenWindows 3.x

It is assumed that you are familiar with the XView windows system from the *user* point of view—that is, you know the appearance and function of the windows, scrollbars, menus, and so forth.

It is also assumed that you are familiar with XView from a *programmer's* point of view, as described in the *XView Programming Manual*. See "Reference" for how to order this book.

B.1 XView Overview

The XView Application Programmer's Interface is an object-oriented, server-based, user-interface toolkit for the X Window System Version 11 (X11). It is designed and documented to help C programmers manipulate XView windows and other XView objects.

Tools

The tool kit is a collection of functions. The runtime system is based on each application having access to a server-based *Notifier*, which distributes input to the appropriate window, and a *window manager* which manages overlapping windows.

There is also a *Selection Service* for exchanging data between windows in the same or different processes.

Objects

XView is an *object-oriented* system. XView applications create and manipulate XView objects. All such objects are associated with XView packages. Objects in the same package share common properties. The major objects are windows, icons, cursors, menus, scrollbars, and frames. A *frame* contains non-overlapping subwindows within its borders.

You manipulate an object by passing a unique identifier or *handle* for that object to procedures associated with that object.

Compatibility

The Solaris 2.x binding is to the 3.2 version of the XView library.

The Solaris 1.x binding is to the 3.0 version of the XView library.

Any new entry points introduced in XView version 3.3 and 3.4 are not supported by our binding.

B.2 FORTRAN 77 Interface

This chapter focuses on manipulating XView windows and objects with FORTRAN 77. The FORTRAN 77 XView interface consists of a set of header files and an interface library. To write XView applications, you need to use the library, header files, object handles, and standard procedures.

Compiling

The library Fxview provides a FORTRAN 77 interface to XView. The actual XView procedures are in the libraries, xview and X11. To compile an XView program, you include some header files and link the libraries.

Example: Compiling an XView program in *Solaris 2.x*:

```
demo% f77 -U -Nx2000 -Nn4000 -o app \
 -I/opt/SUNWspro/SC4.0/include/f77 app.F \
 -lFxview -lxview -lolgx -lX11
```

Above, if you use pixrect, you must put in -lpixrect before -xvol.

Example: Compiling an XView program in *Solaris 1.x*:

```
demo% f77 -U -Nx2000 -Nn4000 -o app \
 -I/usr/lang/SC4.0/include/f77 app.F \
 -lFxview -lxview -lolgx -lX11
```

Initializing

Initialize the XView library using the xv_init function. Some of the functions require special initialization, and some do not, so, in general, it is safer to do this initialization. There are two special aspects:

- This initialization must be done before the FORTRAN 77 main program starts executing its internal initialization code.
- There is a special function named f77_init that each FORTRAN 77 main program always calls before executing its internal initialization code. The version of f77_init provided in libF77 does nothing, but you can provide a substitute version to do the initialization.

The following example shows one way to use `f77_init` to invoke `xv_init`:

```
demo% cat xvini.c
#include <xview/xview.h>
void
f77_init(int *argcp, char ***argvp, char ***envp)
{
    xv_init(XV_INIT_ARGC_PTR_ARGV, argcp, *argvp) ;
    f77_no_handlers = 1 ;                    /* See next paragraph. */
}
demo% cc -o xvini xvini.c
demo% f77 xvini.o Any.f
```

The global variable, `f77_no_handlers`, is a flag that affects subsequent initialization routines. If it is nonzero, the FORTRAN 77 runtime system does not set up any signal handlers.

Signal handlers are for dealing with floating-point exceptions, interrupts, bus errors, segmentation violations, illegal instructions, and so forth.

One problem with XView is that many XView programs do their own signal handling. These programs fail if the FORTRAN 77 runtime system sets and uses the normal signal handlers. These normal signal handlers intercept signals, flush the output buffers, and print a descriptive message. If you have two sets of signal handlers in the same program, they interfere with each other.

Header Files

The header files define the necessary data types, constants, and external procedures necessary to write programs, using the XView interface with FORTRAN 77.

Names of Header Files

Every XView program needs the header file `stddefs_F.h` for standard definitions. It must be first.

The names of the header file are the same as the XView C language header files with the `.h` extension changed to the `_F.h` extension. For example, the FORTRAN 77 header file corresponding to the XView file, `panel.h`, is named `panel_F.h`. Other header files are `canvas_F.h`, `text_F.h`, and so forth.

In addition to the header files corresponding to the XView headers, there are three additional ones. They are:

- `stddefs_F.h`

  This file defines some basic types that are used by most of the other FORTRAN 77 XView header files. You must include this file before any other of the FORTRAN 77 XView files.

- `undef_F.h`

  This file is used if you have more than one subroutine in a single file that needs the XView data types. It undefines certain symbols which are used in the header files, so that you can include the header files in multiple subroutines or functions in the same source file.

- `procitf_F.h`

  This header file contains declarations for routines which will generate interface routines for all procedure types, which are passed to XView.

  Some of the features of XView require you to provide a subroutine that is called by XView when certain events happen. Since FORTRAN 77 routines pass arguments differently than C routines. Since XView assumes the C calling conventions, interface routines are needed to map the arguments correctly.

  The input argument is a FORTRAN 77 subroutine. The output is the address of an interface routine that calls the FORTRAN 77 routine with the arguments properly mapped.

Example: Interpose event call:

```
    EXTERNAL event_func, my_repaint
    CALL set_CANVAS_REPAINT_PROC ( canvas,
&       canvas_repaint_itf ( my_repaint ) )
    err = notify_interpose_event_func ( frame,
&       notify_event_itf ( event_func ),
&       NOTIFY_IMMEDIATE )
```

There can be at most 30 different interface procedures of each type.

Usage of Header Files

To use header files with `f77`, do all three of the following:

- Specify `-I/opt/SUNWspro/include/f77`, in the compile command.
- In your source file, insert the following line:

```
#include "stddefs_F.h"
```

Put it in such `include` lines for any other header files that you need.

- Make `.F` the suffix of your source file.

  When you compile a FORTRAN 77 source file that has a `.F` suffix, the C preprocessor replaces the `#include` line with the contents of the file.

Generic Procedures

There is one general initialization procedure: `xv_init()`.

There are three the standard generic procedures for you to work with objects:

- `xv_create()`
- `xv_find()`
- `xv_destroy()`

Some special procedures for FORTRAN 77 are also available. See "Attribute Lists" for details.

Attribute Procedures

Each class of objects has its own set of *attributes*. Each attribute has a predefined, or default, value. For example, for the class of scrollbars, there is a width and a color.

The standard C interface to XView defines two routines, `xv_get()` and `xv_set()`, which locate and set attributes of XView objects. These routines take an arbitrary number and type of parameters and return various types, depending on its arguments.

Instead of these routines, the FORTRAN 77 interface to XView defines a separate routine to get and set each attribute:

- get—The routine to get the value of an attribute is named get_attrname:
  - Each get routine is a function.
  - Each get routine takes an XView object as the first argument.
  - Each get routine returns the value of the attribute requested.

  For example:

```
    CALL set_WIN_SHOW ( frame, TRUE )
    width = get_CANVAS_WIDTH ( canvas )
```

- set—The routine to set an attribute is named set_attrname:
  - Each set routine is a subroutine.
  - Each set routine takes, as its first argument, the object for which the attribute is being set.
  - The second argument is the value of the attribute.

Attribute Lists

Some of the XView C routines may optionally take extra arguments that are lists of attributes and values. The extra arguments vary in number and type.

The FORTRAN 77 versions of these routines do *not* support this variable number of arguments, and these versions ignore any arguments after the required ones. However, a 0 must be passed as the last argument to be compatible with future versions which may support the extra arguments.

Instead, special versions of these routines are provided that take as a last argument an argument of type Attr_avlist. This type is a pointer to an array of attributes and values. The special routines are:

- xv_init_l()
- xv_create_l()
- xv_find_l()
- selection_ask_l()
- selection_init_request_l()

Example calls:

```
    mymenu = xv_create ( NULL, MENU, 0 )
    ncols = get_MENU_NCOLS ( mymenu )
    call set_MENU_NITEMS ( mymenu, items )
    call xv_find_l ( mymenu, MENU,
&       attr_create_list_2s ( MENU_DEFAULT,4) )
    call xv_destroy( mymenu )
```

Above, `mymenu` is an object of type `XV_object`.

The lists for `Attr_avlist` are created by functions which have the names:

- `attr_create_list_n()`
- `attr_create_list_ns()`

The *n* indicates the number of arguments the routine accepts.

The number of arguments can be 1-16.

The routines ending in `s` return a pointer to a static attribute-value array, which is reused with each call to the static list routines.

The versions without the `s` return a dynamically allocated list, which you pass to `xv_destroy()` when you are finished with the list.

For any attribute which expects a pointer, you must pass `loc()` of the address of the object, instead of the address of the object, because these routines know that FORTRAN 77 passes arguments by reference, and always dereference each argument. This usage is shown in the last example in this appendix.

Handles

If you create an XView object, then `xv_create()` returns a *handle* for that object. You pass this handle to the appropriate procedure for manipulating the object.

Data Types

Each XView object has its own specific data type. The name of an object's data type always starts with a capital letter. For example, the data type for a scrollbar is `Scrollbar`. The standard list of these types is in the header files.

Code Fragment

Here, we provide an example program to illustrate the style of programming with the XView interface in FORTRAN 77. It performs three functions:

- Creates a scrollbar with a page length of 100 units and a starting offset of 10.
- Changes the page length to 20.
- Destroys the scrollbar.

Here is the program:

```
      Scrollbar bar
      bar = xv_create_l ( 0, SCROLLBAR,
 &        attr_create_list_4s ( SCROLLBAR_PAGE_LENGTH, 100,
 &                                SCROLLBAR_VIEW_START, 10 )
 &         )
      call set_SCROLLBAR_PAGE_LENGTH ( bar,  20 )
      call xv_destroy ( bar )
```

In this example:

- `bar` is declared to be of type `Scrollbar`.
- `xv_create_l()` is invoked as a *function*.
- `set_SCROLLBAR_PAGE_LENGTH()` is invoked as a *subroutine*.
- `xv_destroy()` is invoked as a *subroutine*.

B.3 C to FORTRAN 77

In converting C to FORTRAN 77, besides the six standard generic procedures, there are approximately 80 other procedures plus hundreds of attributes. These are all documented in the manual, *XView 1.0 Reference Manual: Summary of the XView API*. The problem is that all of the coding is in C.

You can use the following as you find it in the manual, with no change:

- XView procedure names
- XView object names
- XView object data types (except `Boolean`, more on this below)

However, you must make the following changes:

- Any elementary C data type used must be converted to the corresponding FORTRAN 77 data type.
- If a C procedure *returns* something, then it must be invoked in FORTRAN 77 as a *function*; otherwise, it must be invoked as a *subroutine*.
- The XView type `Boolean` must be converted to the FORTRAN 77 type `LOGICAL`.
- Arguments which are declared as *type*`*` have a FORTRAN 77 type of *type*`_ptr`.
- Arguments of type `struct` *str* have a FORTRAN 77 type of `Str`.

This table summarizes the equivalents of C declarations in FORTRAN 77:

Table B-1 C and FORTRAN 77 Declarations

C	FORTRAN 77
`short int x;`	`INTEGER*2 X`
`long int x;`	`INTEGER*4 X`
`int x;`	`INTEGER*4 X`
`long long int x;`	`INTEGER*8 X` *{requires -dbl}*
`char x;`	`BYTE X or LOGICAL*1 X`
`char *x;`	`CHARACTER*`*n* `X` (See Note following this table)
`char x[6];`	`CHARACTER*6 X`
`float x;`	`REAL X`
`double x;`	`DOUBLE PRECISION X`

Note – The C declaration for "*x is a pointer to a character*" is `char*x`. The FORTRAN 77 declaration for "*X is a character string*" is `CHARACTER*`*n* `X`, where *n* can be any size. The `f77` character string itself must be null-terminated, however. These two declarations are equivalent.

In standard FORTRAN 77, variables of type `INTEGER`, `LOGICAL`, and `REAL` use the same amount of memory. Since `LOGICAL*1` or `BYTE` violate such rules, they are not standard FORTRAN 77, and can result in nonportable programs.

If you use a character constant, it is null-terminated automatically. If you use a character variable, you have to terminate it explicitly with a null character, `CHAR(0)`.

Example: Terminate a variable character string with the null character:

```
CHARACTER X*10, Z*20
X = 'abc'
Z = X // CHAR(0)
```

Sample Translation: C Function Returning Something

In the chapter, “XView Procedures and Macros,” in the manual, *XView 1.0 Reference Manual: Summary of the XView API*, is this entry:

```
texsw_insert()
    Inserts characters in buf into textsw at the current insertion
    point. The number of characters actually inserted is returned.
    This will equal buf_len  unless there was a memory allocation
    failure. If there was a failure, it will return 0.
        Textsw_index
        textsw_insert(textsw, buf, buf_len)
            Textsw textsw;
            char buf;
            int buf_len;
```

A translation to FORTRAN 77 is:

- Leave the object data type `Textsw` as is.
- Since it returns the number of characters inserted, invoke it as a function.

```
Textsw textsw
CHARACTER*4 buf
INTEGER*4 N, buf_len, textsw_insert
…
buf(4:4) = CHAR(0)
N = textsw_insert(textsw, buf, buf_len)
IF ( N .EQ. 0 ) …
```

Sample Translation: C Function Returning Nothing

In the same manual and chapter is this entry:

```
frame_set_rect()
    sets the rect of the frame. X, y is the upper left corner of the
    window coordinate space. Width and height include the window decoration.
        void
        frame_set_rect(frame,rect)
        Frameframe;
        Rectrect;
```

A translation to `f77` is: it does not return anything; invoke as a subroutine.

```
    …
    Frame frame
    Rect rect
    …
    CALL frame_set_rect ( frame, rect )
    …
```

B.4 Sample Programs

Some of the XView C routines (such as `xv_create()`) take a variable number of arguments. The corresponding FORTRAN 77 versions do not. They ignore any arguments passed after the required arguments.

Alternate versions of the variable-argument-list routines are provided with an `_l` appended to the name. The final argument is an attribute-value list which can be created by the `attr_create_list_*` routines.

Sample 1: Hello World—a small `f77` program using the XView toolkit:

```
demo% cat  xhello.F
    PROGRAM hello1F
#include "stddefs_F.h"
#include "frame_F.h"
#include "panel_F.h"
#include "window_F.h"
#include "attrgetset_F.h"
    EXTERNAL loc
    Frame base_frame                    !  Three special  XView type statements
    Panel panel
    Xv_panel_or_item_ptr pi
    base_frame = xv_create ( 0, FRAME, 0 )
    panel = xv_create_l ( base_frame, PANEL, 0 )
    pi = xv_create_l ( panel, PANEL_MESSAGE,
&       attr_create_list_2s ( PANEL_LABEL_STRING,
&                             loc("Hello world!"))
&       )
    CALL window_main_loop ( base_frame )
    END
demo%
```

Compile in *Solaris 2.x*:

```
demo% f77 -U -Nn5000 -Nx2000 -o hello_world \
        xhello.F -lFxview \
         -I/opt/SUNWspro/SC4.0/include/f77 \
         -lxview -lolgx -lX11
xhello.F:
    MAIN xhello:
demo%
```

Compile in *Solaris 1.x*:

```
demo% f77 -U -Nn5000 -Nx2000 -o hello_world \
       xhello.F -lFxview \
        -I/usr/lang/SC4.0/include/f77 \
        -lxview -lolgx -lX11
xhello.F:
   MAIN xhello:
demo%
```

Many warning messages are produced about names being over 32 characters. To suppress these messages, compiled with the `–w` option. Then run the executable file:

```
demo% hello_world
```

Soon after the executable is run, the window pops up as a single frame, with the words, `hello world`, in the frame header.

Sample 2: Create a `tty` subwindow and run `/bin/ls` in it:

```
#include "stddefs_F.h"
#include <textsw_F.h>
#include <frame_F.h>
#include <panel_F.h>
#include <termsw_F.h>
#include <tty_F.h>
        Frame           frame
        character*8     command
        Termsw          tty
        integer         my_argv(2)
        command = '/bin/ls' // char(0)
        my_argv(1) = loc(command)
        my_argv(2) = 0
        call xv_init(0)
        frame = xv_create(0, FRAME, 0)
        tty = xv_create_l(frame, TERMSW,
     1          attr_create_list_2(TTY_ARGV, loc(my_argv)), 0)
        call set_TERMSW_MODE(tty, TTYSW_MODE_TYPE)
        call set_WIN_ROWS(tty, 24)
        call set_WIN_COLUMNS(tty, 80)
        call window_fit(frame)
        call window_main_loop( frame )
        end
```

We pass `loc(my_argv)` to `attr_create_list_2()`.

B.5 Reference

A comprehensive programmer's reference manual, *XView Programming Manual*, is now available from O'Reilly & Associates, Incorporated, as Volume Seven of their series of *X Window System* documentation. To order, contact:

O'Reilly & Associates, Inc.
632 Petaluma Avenue
Sebastopol, CA 95472
Phone: (800) 338-6887
Email: uunet!ora!xview

iMPact: Multiple Processors C

This appendix is organized into the following sections:

This appendix describes ways to spread a set of programming instructions over a multiprocessor system so they execute in parallel. This process is called *parallelizing*; the goal is speed of execution.

C.1 Overview

In general, this compiler can parallelize certain kinds of loops that include arrays. You can let the compiler determine which loops to parallelize, a process called *automatic* parallelizing; or you can specify each loop yourself, known as *explicit* parallelizing.

The parallelizer is integrated tightly with optimization, and operates on the same intermediate representation used by the optimizer.

It is assumed that you are familiar with the concepts of parallel processing, as well as this FORTRAN 77 compiler and the Solaris or UNIX operating system.

Requirements

Multiprocessor FORTRAN 77 requires the following components:

- A Sun multiprocessor system, such as a SPARCstation 10 or 1000 server
- The Solaris 2.3 operating environment or later, that supports multithreading
- iMPact FORTRAN 77 MP

The multiprocessing system can have more than one processor. Solaris 2.3 includes the SunOS 5.3 operating system, which supports the `libthread` library and runs many processors simultaneously. The Solaris 1.x system does not support `libthread`. FORTRAN 77 MP has features that exploit multiprocessors, using the Solaris 2.3 operating system.

Automatic Parallelization

Automatic parallelization is both fast and safe. To automatically parallelize loops:

- Compile with `-autopar`.

  With this option, the software determines which loops are appropriate to parallelize. For example, to turn on automatic parallelization that does all the appropriate loops:

```
demo% f77 -autopar any.f
```

- Make sure you have set the number of processors.

  See Section C.3, "Number of Processors," for the commands.

- Run the executable.

Explicit Parallelizing

Explicit parallelization can produce extra performance. However, this method has a risk of producing incorrect results.

To explicitly parallelize all user-specified loops:

- Determine which loops are appropriate to parallelize.
- Insert a directive *just before* each loop that you want to parallelize.

- Compile with `-explicitpar`.
- Make sure you have set `PARALLEL` to indicate the number of processors.
- Run the executable and check the results very carefully.

For example, to turn on explicit parallelization that does only the `i` loop:

```
demo% cat t1.f
    ...
C$PAR DOALL
    do i = 1, n! This loop gets parallelized.
        a(i) = b(i) * c(i)
    end do

    do k = 1, m! This loop does not get parallelized.
        x(k) = y(k) * z(k)
    end do

    ...
demo% f77 -explicitpar t1.f
```

`C$PAR DOALL` is explained later in this chapter. See page 400.

The `libthread` *Primitives*

If you do your own multithreaded coding that uses the `libthread` primitives, do *not* use `-autopar` or `-explicitpar`. Either do it all yourself, or let the compiler do it. Conflicts and unexpected results may occur if you and the compiler are both trying to manage threads with the same primitives. See the description for `-mt` in Chapter 2, “The Compiler.”

Parallel Options and the Directives

The following table lists f77 parallel options.

Table C-1 Parallel Options for f77

Options	Syntax
Automatic (*only*)	-autopar
Automatic and Reduction	-autopar -reduction
Explicit (*only*)	-explicitpar
Automatic and Explicit	-parallel
Automatic and Reduction and Explicit	-parallel -reduction
Show which loops are parallelized	-loopinfo
Show warnings with explicit	-vpara
Use Sun-style MP directives	-mp=sun
Use Cray- style MP directives	-mp=cray

The following table lists f77 parallel directives.

Table C-2 Parallel Directives for f77

Parallel Directives	Purpose
C$PAR DOALL *optional qualifiers*	Parallelize next loop, if possible
C$PAR DOSERIAL	Inhibit parallelization of next loop
C$PAR DOSERIAL*	Inhibit parallelization of loop nest

Rules and Restrictions for Parallelization

Here is a summary of the rules and restrictions:

- -reduction requires -autopar.
- -autopar includes dependence analysis and loop structure optimization.
- -parallel is equivalent to -autopar -explicitpar.
- -explicit -depend is equivalent to -parallel.
- -noautopar, -noexplicitpar, -noreduction are the negations.
- The parallelization options can be in any order, but must be all lowercase.
- Using a parallel directive has a high risk of nondeterministic results..

- A loop with an explicit directive gets no reductions.

Standards

Multiprocessing is an evolving concept. When standards for multiprocessing are established, the above features may be superceded.

C.2 Speed Gained or Lost

To get faster code from parallelization requires a multiprocessor system; on a single-processor system, the code usually runs slower.

The speed gained with parallelization varies widely with the application. Some programs are inherently parallel and show great speedup. Many have no parallel potential, and show no speedup at all. There is such a wide range of improvement that it is hard to predict what speedup any one program will get.

Variations in Speedups

To illustrate the range of possible speedups, here is a hypothetical scenario.

Assume there are four processors. With parallelization, the following variations occur. The normal upper limit with four processors is about *three times as fast*.

- Many perfectly good programs, tuned for single-processor computation, and with the overhead of the parallelization, *actually run slower.*
- Many programs tuned for single-processor computation, get *no speedup*.
- Some programs run *10% faster.*
- A few less run *50% faster.*
- Even fewer run *100% faster.*
- A few have so much parallelism that they run three or four times faster.

Vectorization Comparison

If you have good speedup on vector machines with an autovectorizing compiler, a first-order rough approximation can be performed as follows:

speedup = vectorization * (number of CPUS -1)

Remember that this is only a first-order rough approximation.

C.3 Number of Processors

To set the number of processors, set the environment variable PARALLEL. The method of setting varies with the shell: csh(1) or sh(1).

In sh:

```
demo$ PARALLEL=4
demo$ export PARALLEL
```

In csh:

```
demo% setenv PARALLEL 4
```

The following are general guidelines, not hard and fast rules. It usually helps to be flexible and experimental with number of processors. Assume that *n* is the number of processors on the machine.

- Do *not* set PARALLEL greater than *n*. Doing so can seriously degrade performance.
- Try PARALLEL set to the number of processors wanted and expected to get.
- In general, allow at least one processor for activities other than the program you are running—for overhead, other users, and so forth.
- For a *one-user*, multiprocessor system, try PARALLEL=*n*-1 and PARALLEL=*n*.
- For a *one-user* system, if the user asks for more processors than are available on the machine, there can be serious degradation in performance.

- For a *multiple-user* system, if all users together ask for more processors than are available on the machine, it can seriously degrade performance.

  If the machine is overloaded with users, it may help to set PARALLEL to much less than *n*. For example, with a 10-user machine, try PARALLEL at 4, 6, or 8. If you ask for 10 and cannot get 10, then you may end up time-sharing some CPUs with other users.

C.4 Automatic Parallelization

This section shows how to automatically parallelize programs for multiple processors. This method is known as *automatic parallelization*.

What You Do

Example: Set the number of processors and parallelize automatically:

```
    do i = 1, 1000          ! ← Parallelized
       a(i) = b(i) * c(i)
    end do
    do k = 3, 1000          ! ← Not parallelized -- dependency
       x(k) = x(k-1) * x(k-2) ! See page 365
    end do
demo% setenv PARALLEL 4             ← Sets the number of processors
demo% f77 -autopar t2.f             ← Tells f77 to parallelize automatically
```

To parallelize automatically:

- Use the -autopar option
- Use the PARALLEL environment variable to set the number of processors

Section C.3, "Number of Processors," shows you how to do so.

To determine which programs benefit from automatic parallelization, study the rules the compiler uses to detect parallelizable constructs. Alternatively, compile the programs with automatic parallelization, then time the executions.

If you do your own multithreaded coding using the `libthread` primitives, do *not* use `-autopar`. Either do it all yourself or let the compiler do it. Conflicts and unexpected results may happen if you and the compiler are both trying to manage threads with the same primitives. See `-mt` in Chapter 2, "The Compiler."

What the Compiler Does

For automatic parallelization, the compiler does two things:

- Dependency analysis to detect loops that are parallelizable
- Parallelization of those loops

What `f77` does for automatic parallelization is similar to the analysis and transformations of a vectorizing compiler.

Loop Parallelization

The compiler applies appropriate dependence-based restructuring transformations. It then distributes the work evenly over the available processors. Each processor executes a different chunk of iterations.

Example: With four CPUs and 1000 iterations, the following are simultaneous:

Processor 1 executing iterations	1	through	250
Processor 2 executing iterations	251	through	500
Processor 3 executing iterations	501	through	750
Processor 4 executing iterations	751	through	1000

Dependency Analysis

A set of operations can be executed in parallel only if the computational result does not depend on the order of execution. The compiler does a dependency analysis to detect loops with no order dependence. If it errs, it does so on the side of caution. Also, it may not parallelize a loop that could be parallelized because the gain in performance does not justify the overhead.

Example: *Automatic* parallelizing skips this loop; it has data dependencies:

```
do k = 3, 1000
    x(k) = x(k-1) * x(k-2)
end do
```

You cannot calculate `x(k)` until two previous elements are ready.

Definitions: Array, Scalar, and Pure Scalar

An *array* variable is one that is declared with dimensioning in a `DIMENSION` statement or a type statement (see the examples).

A *scalar* variable is a variable that is not an array variable.

A *pure scalar* variable is a scalar variable that is not aliased—not referenced in an `equivalence` statement and not in a `pointer` statement.

Examples: Array / scalar—both `m` and `a` are array variables; `s` is pure scalar:

```
dimension a(10)
real m(100,10), s, u, x, z
equivalence ( u, z )
pointer ( px, x )
s = 0.0
...
```

The variables `u`, `x`, `z`, and `px` are scalar variables, but not pure scalar.

Automatic Parallelization Criteria

Automatic parallelization parallelizes `DO` loops that have no inter-iteration data dependencies. This compiler finds and parallelizes any loop that meets the following criteria:

- The construct is a `DO` loop which uses the `DO` statement, but not `DO WHILE`.
- The values of *array* variables for each iteration of the loop do not depend on the values of *array* variables for any other iteration of the loop.

- Calculations within the loop do not *conditionally* change any pure scalar variable that is referenced after the loop terminates.
- Calculations within the loop do not change a *scalar* variable across iterations. This is called *loop-carried dependency.*

There are slight differences from vendor to vendor, since no two vendors have compilers with precisely the same criteria.

Note the exceptions that are described in "Exceptions for Automatic Parallelizing."

Example: Using the `-autopar` option:

```
    ...
    do i = 1, n           ! ← Parallelized
       a(i) = b(i) * c(i)
    end do
    ...
demo% f77 -autopar t.f
```

Apparent Dependencies

Sometimes, the dependencies are only apparent and can be eliminated automatically by the compiler. One of the many transformations the compiler does is make and use its own private versions of some of the arrays. Typically, it can do this if it can determine that such arrays are used in the original loops only as temporary storage.

Example: Using `-autopar`, with dependencies eliminated by private arrays:

`f77` automatically eliminates the apparent dependencies here → and here → by making and using its own private versions of array `a()`.

```
    parameter (n=1000)
    real a(n), b(n), c(n,n)
    do i = 1, 1000          ! ← Parallelized
      do k = 1, n
        a(k) = b(k) + 2.0
      end do
      do j = 1, n
        c(i,j) = a(j) + 2.3
      end do
    end do
    end
```

In the above example, we do not do both inner and outer loops.

Exceptions for Automatic Parallelizing

For automatic parallelization, the compiler does not parallelize a loop if any of the following conditions occur:

- The DO loop is nested inside another DO loop that is parallelized.
- Flow control allows jumping out of the DO loop.
- A user-level subprogram is invoked inside the loop.
- An I/O statement is in the loop.
- Calculations within the loop change an aliased scalar variable.

Nested Loops

Traditionally, both hand and automatic transformations concentrated on the innermost loop, since performance improvements are multiplied by the number of times the outer loops are executed. For example:

```
do i                    ! 10 seconds   10k iterations
    do j                ! 10 seconds   10k iterations
        do k            ! 10 seconds   10k iterations
        end do
    end do
end do
```

On a single processing system, improving the k loop by three seconds results in the performance being increased considerably more than the i loop.

However, on a parallel processing system with a relatively small number of processors, it can be most effective to parallelize the outermost loop. Parallel processing typically involves relatively large loop overheads, so by parallelizing the outermost loop, we minimize the overhead and maximize the work done for each processor.

In general, if there are enough processors, you may want to allocate them *from the top down*. There are many allocation heuristics, some much more complicated than this. The best heuristic requires information about the number of processors, costs of synchronizing parallel threads, and the specific program behavior.

Examples

The following examples illustrate the definition of what is done with automatic parallelization, plus the exceptions.

Example: Using `-autopar`, a *call* inside a loop:

```
    ...
    do 40 kb = 1, n          ! ← Not parallelized
       k = n + 1 - kb
       b(k) = b(k)/a(k,k)
       t = -b(k)
       call daxpy(k-1,t,a(1,k),1,b(1),1)
40  continue
    ...
```

Example: Using `-autopar`, a *constant* step size loop:

```
    parameter (del = 2)
    ...
    do k = 3, 1000, del         ! ← Parallelized
       x(k) = x(k) * z(k,k)
    end do
    ...
```

Example: Using `-autopar`, *nested* loops:

```
    do 900 i = 1, 1000          ! ← Parallelized (outer loop)
       do 200 j = 1, 1000       ! ← Not parallelized (inner loop)
       ...
200    continue
900 continue
```

Example: Using `-autopar`, a *jump* out of a loop:

```
    do i = 1, 1000          ! ← Not parallelized
    ...
       if (a(i) .gt. min_threshold ) go to 20
    ...
    end do
20  continue
    ...
```

Example: Using `-autopar`, a loop that conditionally changes a scalar variable referenced after a loop:

```
    ...
    do i = 1, 1000          ! ← Not parallelized
       ...
       if ( whatever ) s = v(i)
    end do
    t(k) = s
    ...
```

Reduction for Automatic Parallelizing

A construct that collapses an array to a scalar is called a *reduction*. Typical reductions are summing the elements of a vector, Σv_i, or multiplying the elements of a vector, $\prod v_i$. A reduction violates the criterion that calculations within a loop not change a scalar variable in a cumulative way across iterations.

Example: The scalar `s` is changed cumulatively with each iteration:

```
    s = 0.0
    do i = 1, 1000
       s = s + v(i)
    end do
    t(k) = s
```

However, for some constructs, if the reduction is the only factor that prevents parallelization, then it is possible to parallelize the construct anyway. Some reductions occur so frequently that it is worthwhile for the compiler to be able to recognize them as special cases and parallelize the constructs.

What You Do

For reduction, use a combination of the options: `-autopar -reduction`.

What the Compiler Does

For reduction, the compiler parallelizes loops that meet the following criteria:

- The programming construct satisfies all the automatic parallelizing rules, except that there is a reduction.
- The reduction is one of the recognized reductions that are described in the next section.

Example: Automatic with reduction, the sum of elements:

```
    s = 0.0
    do i = 1, 1000         ! ← Parallelized
        s = s + v(i)
    end do
    t(k) = s
    ...
demo% f77 -autopar -reduction any.f1
```

Recognized Reductions

The following table lists the reductions that are recognized by f77.

Table C-1 Reductions Recognized by the Compiler

Mathematical Entity	**Key FORTRAN 77 Statements**
Sum of the elements	`s = s + v(i)`
Product of the elements	`s = s * v(i)`
Dot product of two vectors	`s = s + v(i) * u(i)`
Minimum of the elements	`s = amin( s, v(i))`
Maximum of the elements	`s = amax( s, v(i))`

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

AND of nonpositive elements	`b = .true.`
	`do i = 1, n`
	`    if (v(i) .le. 0) b=b .and. v(i)`
	`end do`

Count nonzero elements	`k = 0`
	`do i = 1, n`
	`    if ( v(i) .ne. 0 ) k = k + 1`
	`end do`

Note – Actually, all forms of the MIN and MAX functions are recognized.

Roundoff and Overflow/Underflow for Reductions

Results from reductions with sums or products of floating-point numbers can be indeterminate for the following reasons:

- In distributing the calculations over the several processors, the compiler and the runtime environment determine the order of the calculations.
- The order of calculation affects the sum or product of floating-point numbers, that is, computer floating-point addition and multiplication are not associative. This way, you can get (or not get) roundoff, overflow, or underflow, depending on how you associate the operands. That is, `(X*Y)*Z` and `X*(Y*Z)` may not have the same roundoff, overflow, or underflow.

In some situations, the error is acceptable; in others, it is not, so use reduction with discretion, depending on your application.

Example: Overflow and underflow, *with* and *without* reduction:

```
demo% cat t3.f
       real A(10002), result, MAXFLOAT
       MAXFLOAT = r_max_normal()
       do 10 i = 1 , 10000, 2
           A(i) = MAXFLOAT
           A(i+1) = -MAXFLOAT
10     continue

       A(5001)=-MAXFLOAT
       A(5002)=MAXFLOAT

       do 20 i = 1 ,10002! Add up the array
           RESULT = RESULT + A(i)
20     continue
       write(6,*) RESULT
       end
demo% setenv PARALLEL 2                     {Number of processors is 2}
demo% f77 -silent -autopar t3.f
demo% a.out
   0.                                       {Without reduction, 0. is correct}
demo% f77 -silent -autopar -reduction t3.f
demo% a.out
  Inf                                       {With reduction, Inf. is not correct}
demo%
```

Example: Roundoff: get the sum of 100,000 random numbers between -1 and +1:

```
demo% cat t4.f
    parameter ( n = 100000 )
    double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
    s = d_lcrans ( v, n, lb, ub ) ! Get n random nos. between -1 and +1
    s = 0.0
    do i = 1, n
        s = s + v(i)
    end do
    write(*, '(" s = ", e21.15)') s
    end
demo% f77 -autopar -reduction t4.f
```

Results vary with the number of processors. The following table shows the sum of 100,000 random numbers between -1 and +1.

Number of Processors	Output
1	`s = 0.568582080884714E+02`
2	`s = 0.568582080884722E+02`
3	`s = 0.568582080884721E+02`
4	`s = 0.568582080884724E+02`

In this situation, the roundoff error is acceptable on the order of 10^{-14} for data that is random to begin with. For more information, see the document, "*What Every Computer Scientist Should Know About Floating-point Arithmetic*" by David Goldberg, which is provided in the online book system.

C.5 Explicit Parallelization

This section shows how to specify a loop for parallelization.

What You Do

To parallelize specific loops, do the following:

- Analyze loops to detect those with no order dependence.
- Insert a directive *just before* each loop that you want to be parallelized.
- Compile with the `-explicitpar` or `-parallel` option.
- Make sure the number of processors is set.
- Run the executable and check the results very carefully.

Example: Parallelize the `i` loop:

```
C$PAR DOALL
      do i = 1, n             ! This loop is parallelized.
         a(i) = b(i) * c(i)
      end do
      do k = 1, m             ! This loop is not parallelized.
         x(k) = y(k) * z(k)
      end do
demo% setenv PARALLEL 2
demo% f77 -explicitpar t1.f
```

The directive, `C$PAR DOALL`, is described later, as is setting the number of processors, and both are illustrated in the following example. See also Section C.3, "Number of Processors."

Note – This method can produce executables that run faster, but there is a risk of incorrect results.

If you do your own multithreaded coding using the `libthread` primitives, do *not* use `-explicitpar`. Either do it all yourself, or let the compiler do it. Conflicts and unexpected results may happen if you and the compiler are both trying to manage threads with the same thread primitives. See `-mt` in Chapter 2, "The Compiler."

What the Compiler Does

For explicit parallelization, the compiler parallelizes those loops that you have specified. This process is similar to the transformations of a vectorizing compiler.

The compiler assumes it can apply some appropriate dependence-based restructuring transformations. It then distributes the work over the available processors. Each processor executes a different chunk of iterations.

For example, with 1,000 iterations, `PARALLEL=4`, and static scheduling, these calculations can be simultaneous:

- Processor 1 executes iterations 1 through 250.
- Processor 2 executes iterations 251 through 500.
- Processor 3 executes iterations 501 through 750.
- Processor 4 executes iterations 751 through 1,000.

Parallel Directive Syntax

A *parallel* directive is a special kind of comment that directs the compiler to parallelize or not to parallelize the specified `DO` loop. Directives are sometimes called pragmas.

A parallel directive consists of one or more *directive lines*.

A directive line is defined as follows:

- The letters of a directive line can be in uppercase, lowercase, or mixed.
- The first 5 characters are `C$PAR`, `*$PAR`, or `!$PAR`.
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.
- Directives are listed in columns 7 and beyond.
- Qualifiers, if any, follow directives—on the same line or continuation lines.
- Multiple qualifiers on one line are separated by commas.
- Spaces before, after, or within a directive or qualifier are ignored.
- Columns beyond 72 are ignored. See `-e` on page 46.

See “Alternate Syntax for Directives” on page 397.

The parallel directives and their purposes are as follows:

Directive	Purpose
DOALL	Parallelize the next loop found, if possible.
DOSERIAL	Do not parallelize the next loop.
DOSERIAL*	Do not parallelize the next nest of loops.

Examples: Some parallel directives with and without qualifiers:

```
C$PAR DOALL                                      No qualifiers

C$PAR DOSERIAL

C$PAR DOALL SHARED(I,K,X,V), PRIVATE(A)          This one-line directive is equivalent to
                                                 the three-line directive that follows.
C$PAR DOALL
C$PAR& SHARED(I,K,X,V)
C$PAR& PRIVATE(A)
```

DOALL *Loop*

A DOALL loop is defined as follows:

- The construct is a DO loop, which uses the DO statement, but not DO WHILE.
- The values of array variables for each iteration of the loop do not depend on the values of array variables for any other iteration of the loop.
- If the loop changes a scalar, then that scalar is not referenced after the loop terminates. Such scalar variables are not guaranteed to have a defined value after the loop terminates, since the compiler does not automatically ensure a proper storeback for them.
- For each iteration, any subprogram that is invoked inside the loop does not reference or change values of *array* variables for any other iteration.
- The DO loop index must be an integer.

Scoping Rules

By definition, a private variable or array is one that is private to a *single iteration* of a loop. The value assigned to a private variable or array in one iteration is not propagated to any other iteration of the loop.

By definition, a shared variable or array is one that is shared with all other iterations. The value assigned to a shared variable or array in an iteration is seen by other iterations of the loop.

If an explicitly parallelized loop contains shared references, then you must ensure that sharing does not cause correctness problems. The compiler does no synchronization on updates or accesses to shared variables.

If you specify a variable as private, and its only initialization is *outside* the loop, then the value of that variable can be undefined in the loop.

Default Scoping Rules for Sun-Style Directives

For Sun-style explicit directives, the compiler uses default rules to determine whether a scalar or array is shared or private. You can override the default rules to specify the attributes of scalars or arrays referenced inside a loop.

The compiler applies these default rules:

- All *scalars* are treated as *private*. A processor local copy of the scalar is made in each processor, and that local copy is used within that process.
- All *array* references are treated as *shared* references. Any write of an array element by one processor is visible to all processors. No synchronization is performed on accesses to shared variables.

If inter-iteration dependencies exist in a loop, then the execution may result in erroneous results. You must ensure that these cases do not arise. The compiler may sometimes be able to detect such a situation at compile-time, and issue a warning, but it does not disable parallelization of such loops.

Example: Potential problem through equivalence:

```
   equivalence (a(1),y)
C$PAR DOALL
   do i = 1,n
      y = i
      a(i) = y
   end do
```

In the above example, since the scalar variable `y` has been equivalenced to `a(1)`, it is no longer a private variable, even though the compiler treats it as such by the default scoping rule. Thus, the presence of the `DOALL` directive may lead to erroneous results when the parallelized `i` loop is executed.

You can alter the above example by using `C$PAR DOALL SHARED(y)`.

`DOALL` *Directive*

Explicit parallelization of a `DOALL` loop requires far more analysis and sophistication than automatic parallelization. There is far more risk of indeterminate results—not only roundoff, but also inter-iteration interference.

To explicitly parallelize a `DOALL` loop, insert a `DOALL` parallel directive immediately before the *specific* loop, and compile with `-explicitpar`.

Note – A loop with an explicit directive does not get automatic reductions.

Example: Explicit parallelization of a `DOALL` loop:

```
demo% cat t4.f
   ...
C$PAR DOALL
   do i = 1, n          ! ← Parallelized
      a(i) = b(i) * c(i)
   end do
   do k = 1, m          ! ← Not parallelized
      x(k) = x(k) * z(k,k)
   end do
   ...
demo% f77 -explicitpar t4.f
```

Example: Explicit parallelization of DOALL; some calls can create dependencies:

```
demo% cat t5.f
    ...
C$PAR DOALL
    do 40 kb = 1, n           ! ← Parallelized
        k = n + 1 - kb
        b(k) = b(k)/a(k,k)
        t = -b(k)
        call daxpy(k-1,t,a(1,k),1,b(1),1)
40  continue
    ...
demo% f77 -explicitpar t5.f
```

This example is an instance where explicit parallelization is useful over automatic parallelization. The code is taken from `linpack`. The subroutine, `daxpy`, was analyzed by a software engineer for iteration dependencies, and found to *not* have any. It is a nontrivial analysis.

`CALL` *in a Loop*

It is sometimes difficult to determine if there are any inter-iteration dependencies. A subprogram invoked from within the loop requires advanced dependency analysis. Since such a case works only under explicit parallelization, *you* must do the advanced dependency analysis, not the compiler.

The following rule sometimes helps with subprogram calls in a loop:

> Within a subprogram, if all local variables are *automatic*, rather than *static*, then the subprogram does not have iteration dependencies.

Even though the above rule is sufficient, it is by no means necessary. For instance, the `daxpy()` routine in the previous example does not satisfy this rule, and it does not have iteration dependencies, although that is not obvious.

You can make all *local* variables of a subprogram automatic in two ways:

- List them in an `automatic` statement. However, then you cannot initialize them in a `data` statement.
- Compile the subprogram with the `-stackvar` option. Doing so can result in stack overflow.

Qualifiers

All qualifiers are optional. The following table summarizes available qualifiers.

Table C-2 `DOALL` Qualifiers

Qualifiers	Action	Syntax
`PRIVATE`	Do not share variables *u1*, *u2*, … between iterations.	`DOALL PRIVATE(` *u1*`,` *u2*`,` … `)`
`SHARED`	Share variables *v1*, *v2*, … between iterations.	`DOALL SHARED(` *v1*`,` *v2*`,` … `)`
`MAXCPUS`	Use no more than *n* CPUs.	`DOALL MAXCPUS(` *n* `)`
`READONLY`	The listed variables are *not* modified in the `DOALL` loop.	`DOALL READONLY(` *v1*`,` *v2*`,` … `)`
`SAVELAST`	Save the values of all *private* variables from the last `DO` iteration.	`DOALL SAVELAST`
`STOREBACK`	Save the values of variables *v1*, *v2*, … from the last `DO` iteration.	`DOALL STOREBACK(` *v1*`,` *v2*`,` … `)`
`REDUCTION`	Treat the variables *v1*, *v2*, … as *reduction* variables for the loop.	`DOALL REDUCTION(` *v1*`,` *v2*`,` … `)`
`SCHEDTYPE`	Set the scheduling type to *t*. (See “SCHEDTYPE(t)” on page 384.)	`DOALL SCHEDTYPE(` *t* `)`

`PRIVATE(`*varlist*`)`

The `PRIVATE(`*varlist*`)` qualifier specifies that all scalars and arrays in the list *varlist* are private for the `DOALL` loop. Both arrays and scalars can be specified as private. In the case of an array, each iteration of the `DOALL` loop gets a copy of the entire array. All other scalars and arrays referenced in the `DOALL` loop, but not contained in the private list, will conform to their appropriate default scoping rules.

Example: Specify a private array:

```
C$PAR DOALL PRIVATE(a)
    do i = 1, n
        a(1) = b(i)
        do j = 2, n
            a(j) = a(j-1) + b(j) * c(j)
        end do
        x(i) = f(a)
    end do
```

In the above example, the array `a` is specified as private to the `i` loop.

SHARED(*varlist*)

The SHARED(*varlist*) qualifier specifies that all scalars and arrays in the list *varlist* are shared for the DOALL loop. Both arrays and scalars can be specified as shared. Shared scalars and arrays are common to all the iterations of a DOALL loop. All other scalars and arrays referenced in the DOALL loop, but not contained in the shared list, will conform to their appropriate default scoping rules.

Example: Specify a shared variable:

```
    equivalence (a(1),y)
C$PAR DOALL SHARED(y)
    do i = 1,n
        a(i) = y
    end do
```

In the above example, the variable `y` has been specified as a variable whose value should be shared among the iterations of the `i` loop.

READONLY(*varlist*)

The READONLY(*varlist*) qualifier specifies that all scalars and arrays in the list *varlist* are read-only for the DOALL loop. Read-only scalars and arrays are a special class of shared scalars and arrays that are not modified in any iteration of the DOALL loop. Specifying scalars and arrays as READONLY indicates to the compiler that it can use a separate copy of that variable or array (with the same value) in each iteration of the DOALL loop.

Example: Specify a read-only variable:

```
    x = 3
C$PAR DOALL SHARED(x),READONLY(x)
    do i = 1, n
        b(i) = x + 1
    end do
```

In the above example, even though the variable `x` is a shared variable, the compiler can still choose to use a separate, private copy of it (with the value 3) in each iteration of the `i` loop because of its `READONLY` specification.

`STOREBACK(`*varlist*`)`

The `STOREBACK(`*varlist*`)` qualifier specifies that all scalars and arrays in the list *varlist* are *storeback* for the `DOALL` loop. A `STOREBACK` variable or array is one whose value is computed in a `DOALL` loop, and this computed value can be used after the termination of the loop. In other words, the last loop iteration values of storeback scalars and arrays may be visible outside of the `DOALL` loop.

Example: Specify the loop index variable as storeback:

```
C$PAR DOALL PRIVATE(x), STOREBACK(x,i)
    do i = 1, n
        x = ...
    end do
    ... = i
    ... = x
```

In the above example, both the variables `x` and `i` are `STOREBACK` variables, even though both variables are private to the `i` loop.

There are some potential problems for `STOREBACK`, however.

The `STOREBACK` operation occurs at the last iteration of the explicitly parallelized loop, regardless if this last iteration is the same as the iteration that last updates the value of the `STOREBACK` variable or array.

Example: STOREBACK variable potentially different from the serial version:

```
C$PAR DOALL PRIVATE(x), STOREBACK(x)
    do i = 1, n
        if (...) then
            x = ...
        end if
    end do
    print *,x
```

In the above example, the value of the STOREBACK variable x that is printed out may not be the same as that printed out by a serial version of the i loop. In the explicitly parallelized case, the processor that processes the last iteration of the i loop (when i = n), which performs the STOREBACK operation for x, may not be the same processor that currently contains the last updated value for x. The compiler issues a warning message on these potential problems.

In an explicitly parallelized loop, arrays are not treated *by default* as STOREBACK, so include them in the list *varlist* if such a storeback operation is desired, for example, if the arrays have been declared as private.

SAVELAST

The SAVELAST qualifier specifies that all *private* scalars and arrays are STOREBACK for the DOALL loop. A STOREBACK variable or array is one whose value is computed in a DOALL loop, and this computed value can be used after the termination of the loop. In other words, the last loop iteration values of STOREBACK scalars and arrays may be visible outside of the DOALL loop.

Example: Specify SAVELAST:

```
C$PAR DOALL PRIVATE(x,y), SAVELAST
    do i = 1, n
        x = ...
        y = ...
    end do
    ... = i
    ... = x
    ... = y
```

In the above example, variables x, y and i are STOREBACK variables.

REDUCTION(*varlist*)

The REDUCTION(*varlist)* qualifier specifies that all variables in the list *varlist* are *reduction* variables for the DOALL loop. A *reduction* variable is one whose partial values can be individually computed on various processors, and whose final value can be computed from all its partial values.

The presence of a list of reduction variables can aid the compiler in identifying that the DOALL loop in question is a reduction loop, and in generating parallel reduction code for it.

Example: Specify a reduction variable:

```
C$PAR DOALL REDUCTION(x)
    do i = 1, n
        x = x + a(i)
    end do
```

In the above example, the variable x is a (*sum*) reduction variable; the i loop is a (*sum*) reduction loop.

SCHEDTYPE(*t*)

The SCHEDTYPE(*t*) qualifier specifies that the specific scheduling type *t* be used to schedule the DOALL loop.

For Sun-style directives, the SCHEDTYPE qualifier has a specific scheduling type, for example, C$PAR& SCHEDTYPE(STATIC).

Scheduling Type	**Action**
STATIC	Use *static* scheduling for this DO loop. Distribute all iterations uniformly to all available processors.
SELF[(*chunksize*)]	Use *self*-scheduling for this DO loop. Distribute *chunksize* iterations to each available processor: • Repeat with the remaining iterations until all the iterations have been processed. • If *chunksize* is not provided, f77 selects a value. Example: With 1000 iterations and *chunksize* of 4, distribute 4 iterations to each CPU.

Scheduling Type	Action
`FACTORING[( m )]`	Use *factoring* scheduling for this `DO` loop. If there are *i* iterations at the start, then distribute *i/(2m)* iterations uniformly to each processor: • Repeat with the remaining iterations until all iterations have been processed. • The number of iterations assigned to each CPU must be at least *m*. • There can be one final smaller residual chunk. • If *m* is not provided, `f77` selects a value. Example: With 1000 iterations and `FACTORING`(4), and 4 CPUs, distribute 125 iterations to each CPU, then 62 iterations, then 31 iterations, …
`GSS[( m )]`	Use *guided self-scheduling* for this `DO` loop. If there are *i* iterations to start with, and *k* CPUs, then: • Assign *i/k* iterations to the first processor. • Assign (*i*-*i*/*k*)/*k* (remaining iterations divided by *k)* to the second processor. • Continue for the remaining iterations, dividing by *k*, assigning to the next processor, until all the iterations have been processed. Note: • The number of iterations assigned to each CPU must be at least *m*. • There can be one final smaller residual chunk. • If *m* is not provided, `f77` selects a value. Example: With 1000 iterations and `GSS`(10), and 4 CPUs, distribute 250 iterations to the first CPU, then 187 to the second CPU, then 140 to the third CPU, …

Multiple Qualifiers

The qualifiers can appear multiple times, with cumulative effect. In case of conflicting qualifiers, the compiler issues a warning message, and the qualifier appearing last prevails.

Example: A three-line directive:

```
C$PAR DOALL MAXCPUS(4) READONLY(S) PRIVATE(A,B,X) MAXCPUS(2)
C$PAR DOALL SHARED(B,X,Y) PRIVATE(Y,Z)
C$PAR DOALL READONLY(T)
```

Example: A one-line equivalent of the above three lines:

```
C$PAR DOALL MAXCPUS(2), PRIVATE(A,Y,Z), SHARED(B,X), READONLY(S,T)
```

`DOSERIAL` *Directive*

The `DOSERIAL` directive tells `f77` *not* to parallelize the specified loop. It applies to the one loop immediately following it, and only if you compile with `-explicitpar` or `-parallel`.

Example: Exclude one loop from parallelization:

```
    do i = 1, n
C$PAR DOSERIAL
        do j = 1, n
            do k = 1, n
                ...
            end do
        end do
    end do
```

In the above example, the `j` loop is not parallelized, but the `i` or `k` loop can be.

`DOSERIAL*` *Directive*

The `DOSERIAL*` directive tells `f77` not to parallelize the specified nest of loops. It applies to the whole nest of loops immediately following it, and only if you compile with `-explicitpar` or `-parallel`.

Example: Exclude a whole nest of loops from parallelization:

```
    do i = 1, n
C$PAR DOSERIAL*
        do j = 1, n
            do k = 1, n
                ...
            end do
        end do
    end do
```

In the above loops, the `j` and `k` loops are not parallelized; the `i` loop may be.

Interaction between `DOSERIAL*` *and* `DOALL`

If both `DOSERIAL` and `DOALL` are specified, the last one prevails.

Example: Specifying both `DOSERIAL` and `DOALL`:

```
C$PAR DOSERIAL*
    do i = 1, 1000
C$PAR DOALL
        do j = 1, 1000
            ...
        end do
    end do
```

In the above example, the `i` loop is not parallelized, but the `j` loop is.

Also, the scope of the `DOSERIAL*` directive does not extend beyond the textual loop nest immediately following it. It is limited to the same function or subroutine that it is in.

Example: DOSERIAL* does not extend to a loop of a called subroutine:

```
      program caller
      common /block/ a(10,10)
C$PAR DOSERIAL*
      do i = 1, 10
          call callee(i)
      end do
      end

      subroutine callee(k)
      common /block/a(10,10)
      do j = 1, 10
          a(j,k) = j + k
      end do
      return
      end
```

In the above example, DOSERIAL* applies only to the i loop and not to the j loop, regardless if the call to the subroutine callee is inlined or not.

Exceptions for Explicit Parallelization

In general, the compiler parallelizes a loop if you explicitly direct it to, but there are exceptions—some loops the compiler just cannot parallelize.

The following are the primary detectable exceptions that may prevent explicitly parallelizing a DO loop. Examples are also included.

- The DO loop is nested inside another DO loop that is parallelized.

  This exception holds for indirect nesting, too. If you explicitly parallelize a loop, and it includes a call to a subroutine, then even if you parallelize loops in that subroutine, still, at runtime, those loops are not run in parallel.
- A flow control statement allows jumping out of the DO loop.
- The index variable of the loop is subject to side effects, such as being equivalenced.

Warning Messages by `-vpara`

If you compile with `-vpara`, you may get a warning message if `f77` detects a problem with explicitly parallelizing a loop. `f77` may still parallelize the loop.

Table C-3 Exceptions that Prevent Explicit Parallelizing

Exception	**Parallelized**	**Message**
Loop is nested inside another loop that is parallelized.	No	No
Loop is in a subroutine, and a call to the subroutine is in a parallelized loop.	No	No
Jumping out of loop is allowed by a flow control statement.	No	Yes
Index variable of loop is subject to side effects.	Yes	No
Some variable in the loop keeps a loop-carried dependency.	Yes	Yes
I/O statement in the loop—*usually unwise, because the order of the output is random.*	Yes	No

Example: Nested loops, not parallelized, no warning:

```
    ...
C$PAR DOALL
    do 900 i = 1, 1000           ! ⟵ Parallelized (outer loop)
        do 200 j = 1, 1000       ! ⟵ Not parallelized, no warning
            ...
200     continue
900 continue
    ...
demo% f77 -explicitpar -vpara t6.f
```

Example: A loop in subroutine; a call to it is in a parallelized loop, which is not parallelized with no warning:

```
C$PAR DOALL
    do 100 i = 1, 200
        ...
        call calc (a, x)
        ...
100 continue
    ...
demo% f77 -explicitpar -vpara t.f
```

↑ *At runtime, loop may run in parallel.*

```
    subroutine calc ( b, y )
    ...
C$PAR DOALL
    do 1 m = 1, 1000
        ...
1   continue
    return
    end
```

↑ *At runtime, loops do not run in parallel.*

In the above example, the loop in the subroutine is not parallelized because the subroutine itself is run in parallel.

Example: Jumping out of loop: not parallelized, with warning:

```
C$PAR DOALL
    do i = 1, 1000          ! ← Not parallelized, with warning
        ...
        if (a(i) .gt. min_threshold ) go to 20
        ...
    end do
20  continue
    ...
demo% f77 -explicitpar -vpara t9.f
```

Example: Index variable subject to side effects: parallelized, no warning:

```
    equivalence ( a(1), y )     ! ← Source of possible side effects
    ...
C$PAR DOALL
    do i = 1, 2000              ! ← Parallelized: no warning, but not safe
        y = i
        a(i) = y
    end do
    ...
demo% f77 -explicitpar -vpara t11.f
```

Example: Variable in loop has loop-carried dependency: parallelized, warning:

```
C$PAR DOALL
    do 100 i = 1, 200                ! ← Parallelized, with warning
        y = y * i                    ! ← y has a loop-carried dependency
        a(i) = y
100 continue
    ...
demo% f77 -explicitpar -vpara t12.f
```

I/O with Explicit Parallelization

You can do I/O in a loop that executes in parallel, provided that:

- It does not matter that the output from different threads is interleaved, so program output is nondeterministic.
- You ensure the safety of executing the loop in parallel, because you must use an explicit directive and the `-explicitpar` or `-parallel` option.

In other words, a loop with I/O is never automatically parallelized. So don't do I/O in loops you want to be considered for automatic parallelization.

Example: I/O statement in loop, parallelized, no warning (*usually unwise*):

```
C$PAR DOALL
    do i = 1, 10          ! ←  Parallelized with no warning (not advisable)
        k = i
        call show ( k )
    end do
    subroutine show( j )
    write(6,1) j
1   format('Line number ', i3, '.')
    end
demo% f77 -silent -explicitpar -vpara t13.f
demo% setenv PARALLEL 2
demo% a.out
(The output displays the numbers 1 through 10, but in a different order each time.)
```

Example: Recursive I/O hangs:

```
    do i = 1, 10          ! ⟵  Parallelized with no warning ---unsafe
       k = i
       print *, list( k ) ! list is a function that does I/O
    end do
    end
    function list( j )
    write(6,"('Line number ', i3, '.')") j
    list = j
    end
demo% f77 -silent -mt t14.f
demo% setenv PARALLEL 2
demo% a.out
```

In the example above, the program deadlocks in `libF77_mt`, and hangs. Press Control-C to regain keyboard control. In general:

- The library `libF77_mt` is MT-safe, but mostly not MT-hot.
- It is not allowed to do recursive (nested) I/O if you compile with `-mt`.

As an informal definition, an interface is *MT-safe* if:

- It can be simultaneously invoked by more than one thread of control
- The caller is not required to do any explicit synchronization before calling the function
- The interface is free of data races

A *data race* occurs when the content of memory is being updated by more than one thread, and that bit of memory is not protected by a lock. In this case, the value of that bit of memory is nondeterministic—the two threads *race* to see who gets to update the thread (but in this case, the one who gets there later, wins!).

An interface is colloquially called *MT hot* if the implementation has been tuned for performance advantage, using the techniques of multithreading. This is not a rigorous definition—one distinction is that MT safe is really meant to be a rigorously defined concept.

For some formal definitions, read *The Solaris 2.4 Multithreaded Programming Guide*. See also the Threads page (The FAQ answers this sort of question): `http://www.sun.com/sunsoft/Developer-products/sig/threads/`

Risk with Explicit Parallelization: Nondeterministic Results

A set of operations can be safely executed in parallel only if the computational result does not depend on the order of execution. For explicit parallelization, *you* (rather than the compiler) specify which constructs to parallelize, and then the compiler parallelizes the specified constructs. That is, you do your own dependency analysis.

If you force parallelization where dependencies are real, then the results depend on the order of execution; they are *nondeterministic*, and you can get incorrect results.

How Testing Fails

An entire test suite can produce correct results over and over again, and then produce incorrect results. What happens is that the number of processors or the system load, or some other parameter changed. So you must test with different numbers of processors, different system loads, and so forth. But this means you cannot be exhaustive in your test cases.

The problem is *not* roundoff, but interference between iterations. An example of this is one iteration referencing an element of an array that is calculated in another iteration, but the reference happens before the calculation.

One approach is systematic analysis of every explicitly parallelized loop. To be sure of correct results, you must be certain there are no dependencies.

Example: Dependency: parallelize explicitly, get *nondeterministic* results:

```
    real a(1001), s / 0.0 /
    do i = 1, 1001! Initialize array a.
       a(i) = i
    end do
C$PAR DOALL
    do i = 1, 1000! This loop has dependencies.
       a(i) = a(i+1)
    end do
    do i = 1, 1000! Get the sum of all a(i).
       s = s + a(i)
    end do
    print *, s! Print the sum.
    end
demo% f77 -explicitpar t1.f
```

In the example above, a different sum (s) probably results every time. Statements like a(i) = a(i+1) are inherently serial in nature.

How Indeterminacy Arises

In a simpler example, with four processors, eight iterations, and the same kind of initialization:

- The first two iterations run on processor 1
- The next two iterations run on processor 2
- …

All processors run simultaneously, and *usually* finish at about the same time. However, the compiler provides no synchronization for arrays, and for many reasons, one processor *can* finish before others; you cannot predict the finishing order in advance.

Processor 1	**Processor 2**	**Processor 3**	**Processor 4**
a(1) = a(2)	a(3) = a(4)	a(5) = a(6)	a(7) = a(8)
a(2) = a(3)	a(4) = a(5)	a(6) = a(7)	a(8) = a(9)

When processor 1 does `a(2) = a(3)`:

- If processor 2 has done `a(3) = a(4)`, then `a(2)` gets `4`.
- If processor 2 has *not* yet done `a(3) = a(4)`, then `a(2)` gets `3`.

Therefore, the values in `a(2)` depend on which processor finishes first. After completion of the parallelized loop, the values in array `a` depend on which processor finishes first and which finishes second, … so the sum depends on events you cannot determine.

The major variables in the runtime environment that cause this kind of trouble are the number of processors in the system, the system load, interrupts, and so forth. However, you usually cannot know them *all*, much less control them all.

Signals

In general, if the loop you are parallelizing does any signal handling, then there is a risk of unpredictable behavior, including system hangs.

In particular, if:

- The I/O statement raises an exception
- The signal handler you provide does I/O

then your system can lock up. These conditions cause problems even on single-processor machines.

Two common ways of doing signal handling without being explicitly aware of it are:

- Input/output statements (`WRITE`, `PRINT`, and so forth) that raise exceptions
- Requesting exception handling

Example: Output that can raise exceptions:

```
real x / 1.0 /, y / 0.0 /
print *, x/y
end
```

Input/output statements do locking, and if an exception is raised then, there may be an attempt to lock an already locked item, resulting in a deadlock.

One possibly over-cautious approach is: if you are parallelizing, do not have I/O in that loop, and do not request exception handling.

Example: Using a signal handler which breaks the rules:

```
      character  string*5, out*20
      double precision value
      external exception_handler
      i = ieee_handler('set', 'all', exception_handler)
      string = '1e310'
      print *, 'Input string ', string, ' becomes: ', value
      print *, 'Value of 1e300 * 1e10 is:', 1e300 * 1e10
      i = ieee_flags('clear', 'exception', 'all', out)
      end

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

The `exception_handler` function is called as a result of the expression, `1e300 * 1e10`, being evaluated in the `print` statement.

The output is:

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

Alternate Syntax for Directives

The following table shows `f77` parallel directive in the Cray style.

Table C-3 Overview of Alternate Directive Syntax for `f77`

Parallel Directive Syntax (Cray Style)

`!MIC$ DOALL`
`!MIC$&  SHARED(` *v1*, *v2*, … `)`
`!MIC$&  PRIVATE(` *u1*, *u2*, … `)`
`...`*optional qualifiers*

Cray Directive Syntax

A parallel directive consists of one or more *directive lines*. A directive line is defined as follows:

- The letters of a directive line can be in uppercase, lowercase, or mixed.
- The first 5 characters are `CMIC$`, `*MIC$`, or `!MIC$`.
- An *initial* directive line has a blank in column 6.
- A *continuation* directive line has a nonblank in column 6.
- Directives are listed in columns 7 and beyond.
- Qualifiers, if any, follow directives—on the same line or continuation lines.
- Multiple qualifiers on a line are separated by commas.
- All variables and arrays are in qualifiers `SHARED` or `PRIVATE`.
- Spaces before, after, or within a directive or qualifier are ignored.

Columns beyond 72 are ignored.

Forms of Parallel Directives

Parallel directives have two forms: Cray style and Sun style. The default is Sun style (`-mp=sun`). If you use Cray-style directives, you must compile with `-mp=cray`.

A program compiled and run with both the Sun and Cray computers can produce different results.

With the Cray style, you must assign each and every scalar and array within the loop to either a `SHARED` or a `PRIVATE` qualifiers.

Qualifiers (Cray Style)

For Cray-style directives, the PRIVATE qualifier is required, and it is not optional. Each variable within the DO loop must be qualified as private or shared, and the DO loop index must always be private. The following table summarizes available Cray-style qualifiers.

Table C-1 DOALL Qualifiers (Cray Style)

Qualifier	**Action**
SHARED(*v1*, *v2*, …)	Share the variables *v1*, *v2*, … between parallel processes. That is, they are accessible to all the tasks.
PRIVATE(*x1*, *x2*, …)	Do not share the variables *x1*, *x2*, … between parallel processes. That is, each task has its own private copy of these variables.
SAVELAST	Save the values of *private* variables from the last DO iteration.
MAXCPUS(*n*)	Use no more than *n* CPUs.

For Cray-style directives, the DOALL directive allows a scheduling qualifier, for example, !MIC$& SINGLE. Use at most one scheduling qualifier for any particular directive.

Table C-2 DOALL Cray Scheduling

Qualifier	**Action**
SINGLE	Distribute *one* iteration to each available processor.
CHUNKSIZE(*n*)	Distribute *n* iterations to each available processor. *n* is an expression. For best performance, *n* must be an integer constant. Example: With 100 iterations and CHUNKSIZE(4), distribute 4 iterations to each CPU.
NUMCHUNKS(*m*)	If there are *i* iterations, then distribute *i/m* iterations to each available processor. There can be one smaller residual chunk. *m* is an expression. For best performance, *m* must be an integer constant. Example: With 100 iterations and NUMCHUNKS(4), distribute 25 iterations to each CPU.
GUIDED	Distribute the iterations by use of guided self-scheduling. This distribution minimizes synchronization overhead, with acceptable dynamic load balancing.

The default scheduling type is the Sun-style STATIC.

C.6 Debugging Tips and Hints for Parallelized Code

The parallelization options limit the utility of debugging the program with `dbx`. Only the `dbx where` command will be enabled, allowing a symbolic traceback of the parallelized program.

While the `-autopar`, `-explicitpar`, and `-parallel` options generate code that conflicts with `-g`, `dbx` can still be used to display a symbolic traceback. However, `dbx` will not be able to display the value of any variables in the parallelized program.

Although the `-g` option does not inhibit parallelization of the program by the `-autopar`, `-explicitpar`, and `-parallel` options, it does reduce the utility of `dbx` in debugging these programs. Some alternative ways to debug parallelized code are suggested below.

Some Solutions without `dbx`

Debugging parallelized programs requires some cleverness. The following schemes suggest ways to approach the problem:

- Turn off parallelization.

  You can do one of the following:

  - Turn off the parallelization options—Compile and run the program first with `-O3` or `-O4`, but without `-autopar`, `-explicitpar`, and `-parallel` to verify that it works correctly.
  - Set the CPUs to one—Run the program with the environment variable, `PARALLEL=1`.

  If the problem disappears, then you know it is due to parallelization.

  If the problem remains, and you are using `-autopar`, then the compiler is parallelizing something it should not. Some differences may exist, because parallelized programs are always optimized.

- Turn off `-reduction`.

  If you are using the `-reduction` option, summation reduction may be occurring and yielding slightly different answers. Try running without this option.

- Reduce the number of compile options.

  Try to reduce the number of compile options to the minimum set of `-parallel -O3` and see if the results. are correct.

- Use `fsplit`.

  If you have a lot of subroutines in your program, use `fsplit` to break them into separate files. Then compile some with and without `-parallel`, and use `ld` to link the `.o` files. You need to use `-parallel` on the `ld` command.

  Execute the binary and verify results.

  Repeat this process until the problem is narrowed down to one subroutine.

  You can proceed with using a dummy subroutine or explicit parallelization to track down the loop that causes the problem.

- Use `-loopinfo`

  Check which loops are being parallelized and which loops are not.

- Use a dummy subroutine

  Create a dummy subroutine or function which does nothing. Put calls to this subroutine in a few of the loops which are being parallelized. Recompile and execute. Use `-loopinfo` to see which loops are being parallelized.

  Continue this process until you start getting the correct results.

  Then remove the calls from the other loops, compile, and execute to verify that you are getting the correct results.

- Use explicit parallelization.

  Add the `C$PAR DOALL` directive to a couple of the loops which are being parallelized. Compile with `-explicitpar`, then execute and verify the results. Use `-loopinfo` to see which loops to get the loops which are being parallelized. This method permits the addition of I/O statements to the parallelized loop.

  Repeat this process until you find the loop that causes the wrong results.

Note – If you need `-explicitpar` only (without `-autopar`), do *not* compile with `-explicitpar` and `-depend`. This method is the same as compiling with `-parallel`, which, of course, includes `-autopar`.

- Run loops *backwards* serially.

  Replace `DO I=1,N` with `DO I=N,1,-1`. Different results point to data dependences.

- Avoid using the loop index.

  It is safer to do so in the loop body, especially if the index is used as an argument in a call.

```
    DO I=1,N                    ! Replace this DO statement

    DO I1=1,N    ! with these two statements.
    I=I1
```

One Solution with `dbx`

To use `dbx` on a parallel loop—temporarily rewrite the program as follows:

- Isolate the body of the loop in a file and subroutine of its own.
- In the original routine, replace loop body with a call to the new subroutine.
- Compile the new subroutine with `-g` and no parallelization options.
- Compile the changed original routine with parallelization and no `-g`.

Example: Manually transform a loop to allow using dbx in parallel:

Original: split `loop.f` into two parts:
Part 1 on `loop1.f`
Part 2 on `loop2.f`

```
demo% cat loop.f
C$PAR DOALL
      DO i = 1,10
          WRITE(0,*) 'Iteration ', i
      END DO
      END
```

Part 1: Loop replaced loop body (the "main")

```
demo% cat loop1.f
C$PAR DOALL
      DO i = 1,10
          k = i
          CALL loop_body ( k )
      END DO
      END
```

Part 2: Body of the loop →

```
demo% cat loop2.f
      SUBROUTINE loop_body ( k )
      WRITE(0,*) 'Iteration ', k
      RETURN
      END
```

Compile Part 1: parallel, no dbx.
Compile Part 2: dbx, no parallel.
Bind both into `a.out`.
Start `a.out` under dbx control.

Put a breakpoint into the loop body.

Run.

dbx stops at the breakpoint.

Show k. See the debugger documentation.

```
demo% f77 -O3 -c -explicitpar loop1.f
demo% f77 -c -g loop2.f
demo% f77 loop1.o loop2.o -explicitpar
demo% dbx a.out            ← Various dbx messages not shown
(dbx) stop in loop_body
(2) stop in loop_body
(dbx) run
Running: a.out
(process id 28163)
t@1 (l@1) stopped in loop_body at line 2 in file "loop2.f"
    2           write(0,*) 'Iteration ', k
(dbx) print k
k = 1                      ← Various values other than 1 are possible
(dbx)
```

Index

Symbols

Numerics

A

B

C

E

F

G

H

I

M

N

P

T

W

X

Z

Join the SunPro SIG Today

Sun Programmer Special Interest Group

The benefits are SIGnificant

At SunSoft, in the Software Development Products business of Sun Microsystems, our goal is to meet the needs of professional software developers by providing the most useful line of software development products for the Solaris platform. We've also recently formed a special interest group, SunPro SIG, designed to provide a worldwide forum for exchanging software development information. This is your invitation to join our world-class organization for professional programmers. For a nominal annual fee of $20, your SunPro SIG membership automatically entitles you to:

- Membership on an International SunPro SIG Email Alias

  Share tips on performance tuning, product feedback, or anything you wish; available as a UUNET address and a dial-up number

- Subscription to the SunProgrammer Quarterly Newsletter

  Includes advice on getting the most out of your code, regular features, guest columns, product previews and the latest industry gossip

- Access to a Repository of Free Software

  SunSoft will collect software related to application development and make it available for downloading

- Free SunSoft Best-of-Repository CD-ROM

  Periodically, we'll take the cream of the crop from the depository and distribute it to members annually

- Free Access to SIG Events

  Including national events, like SIG seminars held at the SUG conference, and regional SunPro SIG seminars

SPECIAL OFFER

Sign up today, and receive a SunPro SIG Tote Bag: A spiffy 15" x 12" black nylon Cordura tote with the SIG logo, proof positive of your Power Programmer status.

So join the SunPro SIG today. And plug into what's happening in SPARC and Solaris development world-wide. Simply complete the form below.

Mail to: SunPro SIG, 2550 Garcia Avenue MS UMPK 03-205, Mountain View, CA,94043-1100

TEL: (415) 688-9862

or

FAX: (415) 968-6396

Unfortunately we cannot accept credit card orders via Email since we need to have your signature on file.

<table>
<tr><td colspan="2">Sign me up for SunPro SIG!
Sun Programmer Special Interest Group</td><td rowspan="12">I'd like to pay for my one-year membership fee of $20 by:

❐ VISA

❐ MASTERCARD

Card # _______________
Expiration Date: _________
Signature:
____
❐ Check made payable to SunSoft</td></tr>
<tr><td colspan="2">Date</td></tr>
<tr><td colspan="2">Name</td></tr>
<tr><td colspan="2">Title</td></tr>
<tr><td colspan="2">Company</td></tr>
<tr><td colspan="2">Email Address</td></tr>
<tr><td colspan="2">Address</td></tr>
<tr><td>City</td><td>State</td></tr>
<tr><td>ZIP</td><td>Country</td></tr>
<tr><td colspan="2">Phone</td></tr>
<tr><td colspan="2">Fax</td></tr>
<tr><td colspan="2">ALL INFO MUST BE FILLED OUT

SunSoft, A Sun Microsystems, Inc. Business</td></tr>
</table>

Adobe PostScript