Fortran Programmer's Guide

Fortran 77 4.2
Fortran 90 1.2

THE NETWORK IS THE COMPUTER®

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5664-10
Revision A, December 1996

Contents

Tables

Preface

This guide combines the essential information programmers need to develop efficient applications using the two Sun™ Fortran compilers, `f77` (Fortran 77 Release 4.2) and `f90` (Fortran 90 Release 1.2). It deals primarily with issues relating to input/output, program development, use and creation of software libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and the C/Fortran interface.

Discussion of the compiler command-line options and their use can be found in the companion book, Sun *Fortran User's Guide.*

Note – This guide covers the Sun Fortran 77 and Fortran 90 compilers. The text uses "`f77`/`f90`" and "Fortran" to indicate information that is common to *both* compilers.

Audience

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

Organization of This Guide

This guide is organized into the following chapters and appendixes:

- **Chapter 1, "Introduction,"** briefly describes the features of the compilers.
- **Chapter 2, "Fortran Input/Output,"** discusses how to use I/O efficiently.
- **Chapter 3, "Program Development,"** demonstrates how program management tools like SCCS, `make`, and Teamware can be helpful.
- **Chapter 4, "Libraries,"** explains use and creation of software libraries.
- **Chapter 5, "Program Analysis and Debugging,"** describes use of `dbx` and other analysis tools.
- **Chapter 6, "Floating Point Arithmetic,"** introduces important issues regarding numerical computation accuracy.
- **Chapter 7, "Porting,"** considers porting programs to Sun compilers.
- **Chapter 8, "Performance Profiling,"** describes techniques for performance measurement.
- **Chapter 9, "Performance and Optimization,"** indicates ways to improve execution performance of Fortran programs.
- **Chapter 10, "Parallelization,"** explains the multiprocessing features of the compilers.
- **Chapter 11, "C-Fortran Interface,"** describes how C and Fortran routines can call each other and pass data.

Multiplatform Release

The Sun Fortran documentation covers the release of the Fortran compilers on a number of operating systems and hardware platforms:

Fortran 77 4.2 is released for:

- Solaris 2.x operating system on:
  - architectures based on the SPARC™ microprocessor
  - x86-based architectures, where x86 refers to the Intel® implementation of one of the following: Intel 80386™, Intel 80486™, Pentium™, or the equivalent
  - PowerPC™ architecture compliant with the Common Hardware Reference Platform (CHRP) and the PowerPC Reference Platform (PReP) specifications

Fortran 90 1.2 is released for:

- Solaris 2.x operating system on SPARC architectures only.

The Fortran documentation describes the Sun compilers on all the above operating systems and platforms. Issues unique to one or more platforms is identified as "*(SPARC)*", "*(Intel)*", "*(PowerPC)*".

Conventions in Text

This manual uses the following conventions to display information.

- Code listings and examples appear 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
```

- Fortran 77 examples appear in tab format, while Fortran 90 examples appear in free format. Examples common to both Fortran 77 and 90 use tab format except where indicated.
- Uppercase characters are generally used to show Fortran keywords and intrinsics (`PRINT`), and lowercase or mixed case is used for variables (`TbarX`).
- The Sun Fortran compilers are referred to by their command names, either `f77` or `f90`. "`f77/f90`" indicates information that is common to both the Fortran 77 and Fortran 90 compilers.

- References to online man pages appear with the topic name and section number. For example, a reference to `GETENV` will appear as `getenv`(3F), implying that the man command to access this page would be:

  ```
  man -s 3F getenv
  ```

Introduction 1

The Sun Fortran compilers, `f77` and `f90`, described in this book (and the companion book Sun *Fortran User's Guide*) are available under the Solaris 2.x operating systems on the various hardware platforms that Solaris supports. The compilers themselves conform to published Fortran language standards, and provide many extended features, including multiprocessor parallelization, sophisticated optimized code compilation, and mixed C/Fortran language support.

Standards Conformance

- `f77` conforms to the ANSI X3.9-1978 Fortran standard and the corresponding International Standards Organization number is ISO 1539-1980. NIST (formerly GSA and NBS) validates it at appropriate intervals.
- `f77` conforms to the standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- `f90` conforms to the ANSI X3.198-1992 standard.
- Both compilers provide an IEEE standard 754-1985 floating-point package.
- On SPARC systems, both compilers provide support for the optimization-exploiting features of SPARC V8, including the SuperSPARC™ implementation. These features are defined in the *SPARC Architecture Manual: Version 8*.

1

Features of the Fortran Compilers

Sun Fortran compilers provide the following features or extensions:

- Global program checking across routines for consistency of arguments, commons, parameters, and the like. (`f77`)
- Support for multiprocessor systems, including automatic and explicit loop parallelization, is integrated tightly with optimization. (*SPARC only*)

Note – Parallelization features of the Fortran compilers are only available with the Sun Performance WorkShop.

- Many VAX/VMS Fortran 5.0 extensions, including (`f77`):
  - `NAMELIST`
  - `DO WHILE`
  - Structures, records, unions, maps
  - Variable format expressions
  - Recursion
  - Pointers
  - Double-precision complex
  - Quadruple-precision real (*SPARC and PowerPC*)
  - Quadruple-precision complex (*SPARC and PowerPC*)
- Cray-style parallelization directives, with extensions on `f90`.
- Global, peephole, and potential parallelization optimizations produce high performance applications. Benchmarks show that optimized applications can run significantly faster when compared to unoptimized code.
- Common calling conventions on Solaris systems permit routines written in C, C++, or Pascal to be combined with Fortran programs.

Related Sun Documentation

The following Sun manuals and guides provide additional information that supplements this book:

- *Fortran 77 4.2 Language Reference.* Complete Fortran 77 reference.
- *Fortran 90 Handbook*. Complete Fortran 90 reference. (Available online with AnswerBook only.)
- *Fortran Library Reference.* Detailed reference to the Sun Fortran 77 and Fortran 90 runtime libraries.
- *Fortran User's Guide*. Complete information on command–line options and how to use the compilers.
- *WorkShop: Command–Line Utilities.* Information on using the `dbx` debugger.
- *WorkShop: Beyond the Basics.* Using the interactive debugger.
- *Numerical Computation Guide.* Details floating-point computation numerical accuracy issues.
- *Linker and Libraries Guide.* Complete information on linking and libraries.
- *Incremental Link Editor.* Using the incremental linker.
- *Performance Profiling Tools.* A guide to the use of performance profiling tools.

1

Fortran Input/Output 2

This chapter discusses the input/output features provided by Sun Fortran compilers. Many of the I/O features found in `f77` are not available in this release (1.2) of `f90`; this chapter primarily describes `f77` features. `f90` is discussed at the end of the chapter.

Accessing Files From Fortran 77 Programs

Data is transferred between the program and devices or files through a Fortran *logical unit*. Logical units are identified in an I/O statement by a logical unit number, a nonnegative integer from 0 to the maximum 4-byte integer value (2,147,483,647).

The character `*` may appear as a logical unit identifier. The asterisk stands for *standard input file* when it appears in a `READ` statement; it stands for *standard output file* when it appears in a `WRITE` or `PRINT` statement.

A Fortran logical unit can be associated with a specific, named file through the `OPEN` statement. Also, certain "preconnected" units are automatically associated with specific files at the start of program execution.

Accessing Named Files

The `OPEN` statement's `FILE=` specifier establishes the association of a logical unit to a named, physical file at runtime. This file may be pre-existing or created by the program. See the Sun *Fortran 77 Language Reference* for a full discussion of the `OPEN` statement.

The FILE= specifier on an OPEN statement may specify a simple file name (FILE='myfile.out') or a file name preceded by an absolute or relative directory path (FILE='../Amber/Qproj/myfile.out'). Also, the specifier may be a character constant, variable, or character expression.

Library routines GETARG(*argnumber*,*charvalue*) and GETENV(*envar*,*charvalue*) can be used to bring command–line arguments and environment variables respectively into the program as character variables that can be used as file names in OPEN statements. (See man page entries for getarg(3F) and getenv(3F) for details).

The following example (GetFilNam.f) shows one way to construct an absolute path file name from a typed-in name. The program uses the library routines GETENV, LNBLNK, and GETCWD to return the value of the $HOME environment variable, find the last non-blank in the string, and determine the current working directory:

```
    CHARACTER F*128, FN*128, FULLNAME*128
    PRINT*, 'ENTER FILE NAME:'
    READ *, F
    FN = FULLNAME( F )
    PRINT *, 'PATH IS: ',FN
    END

    CHARACTER*128 FUNCTION FULLNAME( NAME )
    CHARACTER NAME*(*), PREFIX*128
C       This assumes C shell.
C       Leave absolute path names unchanged.
C       If name starts with '~/', replace tilde with home
C       directory; otherwise prefix relative path name with
C       path to current directory.
    IF ( NAME(1:1) .EQ. '/' ) THEN
        FULLNAME = NAME
    ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
        CALL GETENV( 'HOME', PREFIX )
        FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&               NAME(2:LNBLNK(NAME))
    ELSE
        CALL GETCWD( PREFIX )
        FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
&               '/' // NAME(:LNBLNK(NAME))
    ENDIF
    RETURN
    END
```

Compiling and running `GetFilNam.f` results in:

```
demo% pwd
/home/users/auser/subdir
demo% f77 -silent -o getfil GetFilNam.f
demo% getfil
anyfile
/home/users/auser/subdir/anyfile
demo%
```

Opening Files Without a Name

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

Opened as Scratch

Specifying `STATUS='SCRATCH'` in the `OPEN` statement opens a file with a name of the form `tmp.F`*AAAxnnnnn* — where *nnnnn* is replaced by the current process ID, *AAA* is a string of three characters, and *x* is a letter; the *AAA* and *x* make the file name unique. This file is deleted upon termination of the program or execution of a `CLOSE` statement, unless `STATUS='KEEP'` is specified in the `CLOSE` statement.

Already Open

If the file has already been opened by the program, you can use a subsequent `OPEN` statement to change some of the file's characteristics – specifically, `BLANK` and `FORM`. In this case, you would specify only the file's logical unit number and the parameters to change.

Preconnected Units

Three unit numbers are automatically associated with specific standard I/O files at the start of program execution. These preconnected units are *standard input*, *standard output*, and *standard error*. For Fortran 77:

- Standard input is logical unit 5
- Standard output is logical unit 6

- Standard error is logical unit 0

With Fortran 90, an additional three unit numbers are also preconnected:

- Standard input is logical units 5 and 100
- Standard output is logical units 6 and 101
- Standard error is logical units 0 and 102

Typically, standard input receives input from the workstation keyboard; standard output and standard error display output on the workstation screen.

In all other cases where a logical unit number but no `FILE=` name is specified on an `OPEN` statement, a file is opened with a name of the form `fort.`*n*, where *n* is the logical unit number.

Opening Files Without an `OPEN` *Statement*

Use of the `OPEN` statement is optional in those cases where default conventions can be assumed. If the first operation on a logical unit is an I/O statement other than `OPEN`, the file `fort.`*n* is referenced, where *n* is the logical unit number (except for 0, 5, and 6, which have special meaning).

These files need not exist before program execution. If the first operation on the file is not an `OPEN` or `INQUIRE` statement, they are created.

Example: If the `WRITE` in the code below is the first I/O statement issued on unit 25, the file `fort.25` is created:

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

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

```
demo% f77 -silent -o testunit TestUnit.f
demo% testunit
demo% cat fort.25
  25
demo%
```

Passing File Names to Programs

The file system does not have any automatic facility to associate a logical unit number in a Fortran program with a physical file.

However, there are several satisfactory ways to communicate file names to a Fortran program.

Via Runtime Arguments and GETARG

The library routine `getarg`(3F) can be used to read the command-line arguments at runtime into a character variable. The argument is interpreted as a file name and used in the `OPEN` statement `FILE=` specifier:

```
demo% cat testarg.f
         CHARACTER outfile*40
C  Get first arg as output file name for unit 51
         CALL getarg(1,outfile)
         OPEN(51,FILE=outfile)
         WRITE(51,*) 'Writing to file: ', outfile
         END
demo% f77 -silent -o tstarg testarg.f
demo% tstarg AnyFileName
demo% cat AnyFileName
 Writing to file: AnyFileName
demo%
```

Via Environment Variables and GETENV

Similarly, the library routine getenv(3F) can be used to read the value of any environment variable at runtime into a character variable that in turn is interpreted as a file name:

```
demo% cat testenv.f
         CHARACTER outfile*40
C  Get $OUTFILE as output file name for unit 51
         CALL getenv('OUTFILE',outfile)
         OPEN(51,FILE=outfile)
         WRITE(51,*) 'Writing to file: ', outfile
         END
demo% f77 -silent -o tstenv testenv.f
demo% setenv OUTFILE EnvFileName
demo% tstenv
demo% cat EnvFileName
 Writing to file: EnvFileName
demo%
```

Note that when using getarg or getenv, care should be taken regarding leading or trailing blanks. Additional flexibility to accept relative path names can be programmed along the lines of the FULLNAME function in the example at the beginning of this chapter.

Logical Unit Preattachment Using IOINIT *(*f77 *Only)*

The library routine IOINIT can also be used with f77 to attach logical units to specific files at runtime. IOINIT 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 the Sun *Fortran Library Reference*, and the IOINIT(3F) man page. (*The* IOINIT *facility is not implemented for* f90.)

Example: Associate physical files test.inp and test.out in the current directory to logical 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., 'TST',.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 -o tstinit ini1.f
demo% tstinit
demo% cat ini1.out
  12    3.14159  6
demo%
```

IOINIT is adequate for most programs as written. However, it is written in Fortran specifically to serve as an example for similar user-supplied routines. Retrieve a copy from the following file, a part of the Fortran 77 package installation: /opt/SUNWspro/SC4.2/src/ioinit.f

Command-Line I/O Redirection and Piping

Another way to associate a physical file with a program's logical unit number is by redirecting or piping the preconnected standard I/O files. Redirection or piping occurs on the runtime execution command.

In this way, a program that reads standard input (unit 5) and writes to standard output (unit 6) or standard error (unit 0) can, by redirection (using `<, >, >>, >&, |, |&, 2>, 2>&1` on the command-line), read or write to any other named file. This is shown in Table 2-1:

Table 2-1 `csh`/`sh` Redirection and Piping on the command-line

Action	**Using** `csh`	**Using** `sh`
Standard input — read from `mydata`	`myprog < mydata`	`myprog < mydata`
Standard output — write (overwrite) `myoutput`	`myprog > myoutput`	`myprog > myoutput`
Standard output — write/append to `myoutput`	`myprog >> myoutput`	`myprog >> myoutput`
Pipe standard output to input of another program	`myprog1 | myprog2`	`myprog1 | myprog2`
Pipe standard error and output to another program	`myprog1 |& myprog2`	`myprog1 2>&1 | myprog2`

See the `csh` and `sh` man pages for details on redirection and piping on the command-line.

VAX/VMS Logical File Names (`f77` Only)

If you are porting from VMS FORTRAN to Fortran 77, 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` is set and the `-xl` or `-xld` compiler options are used, 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. Any trailing `/list` or `/nolist` is stripped from the file name in the `INCLUDE` statement. Logical names in a file name are delimited by the first colon in the VMS file name. The compiler converts file names of the form:

```
lname1:file
```

to:

```
path1/file
```

Uppercase and lowercase are significant in logical names. If a logical name is encountered on the `INCLUDE` statement that was not specified by `LOGICALNAMEMAPPING`, the file name is used unchanged.

Direct I/O

Direct or random I/O allows you to access a file directly by record number. Record numbers are assigned when a record is written. Unlike sequential I/O, direct I/O records can be read and written in any order. However, in a direct access file, all records must be the same fixed length. Direct access files are declared with the `ACCESS='DIRECT'` specifier on the `OPEN` statement for the file.

A logical record in a direct access file is a string of bytes of a length specified by the `OPEN` statement's `RCL=` specifier. `READ` and `WRITE` statements must not specify logical records larger than the defined record size. (Record sizes are specified in bytes.) Shorter records are allowed. Unformatted, direct writes leave the unfilled part of the record undefined. Formatted, direct writes cause the unfilled record to be padded with blanks.

Direct access READ and WRITE statements have an extra argument, REC=*n*, to specify the record number to be read or written.

Example: Direct access, *unformatted:*

```
    OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=200,
&       FORM='UNFORMATTED', ERR=90 )
    READ( 2, REC=13, ERR=30 ) X, Y
```

This program opens a file for direct access, unformatted I/O, with a fixed record length of 200 bytes, then reads the thirteenth record into X and Y.

Example: Direct access, *formatted*:

```
    OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=200,
&       FORM='FORMATTED', ERR=90 )
    READ( 2, FMT='(I10,F10.3)', REC=13, ERR=30 ) A, B
```

This program opens a file for direct access, formatted I/O, with a fixed record length of 200 bytes. It then reads the thirteenth record and converts it to the format(I10,F10.3).

For formatted files, the size of the record written is determined by the FORMAT statement. In the preceding example, the FORMAT statement defines a record of 20 characters or bytes. More than one record can be written by a single formatted write if the amount of data on the list is larger than the record size specified in the FORMAT statement. In such a case, each subsequent record is given successive record numbers.

Example: Direct access, formatted, *multiple record write*:

```
    OPEN( 21, ACCESS='DIRECT', RECL=200, FORM='FORMATTED')
    WRITE(21,'(10F10.3)',REC=11) (X(J),J=1,100)
```

The write to direct access unit 21 creates 10 records of 10 elements each (since the format specifies 10 elements per record) these records are numbered 11 through 20.

Internal Files

An internal file is an object of type CHARACTER such as a variable, substring, array, element of an array, or field of a structured record. Internal file READs can be from a *constant* character string. I/O on internal files simulates formatted READ and WRITE by transferring and converting data from one character object to another data object. No physical I/O is actually performed.

When using internal files:

- The name of the character object receiving the data appears in place of the unit number on a WRITE statement. On a READ statement, the name of the character object source appears in place of the unit number.
- A constant, variable, or substring object constitutes a single record in the file.
- With an array object, each array element corresponds to a record.
- 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 *(f77 only).*
- 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 -o tstintern intern1.f
demo% tstintern
 12 99
  12  99
demo%
```

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

```
demo% cat intern2.f
    CHARACTER  LINE(4)*16    ! This is our "internal file"
*                    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  .
    PRINT *, I, J, K, L, M, N
    END
demo% f77 -silent intern2.f
demo% a.out
   81  81  82  82  83  83
demo%
```

Example: Direct access read from an internal file (one record) *(f77 only)*:

```
demo% cat intern3.f
    CHARACTER LINE(4)*16    ! This is our "internal file"
*                    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
20  FORMAT( I4, I4 )
    PRINT *, M, N
    END
demo% f77 -silent intern3.f
demo% a.out
   83  83
demo%
```

Tape I/O

Most typical Fortran I/O is done to disk files. However, by associating a logical unit number to a physically mounted tape drive via the `OPEN` statement, it is possible to do I/O directly to tape.

It is more reliable and efficient to use the `TOPEN()` routines rather than Fortran I/O statements to do I/O on magnetic tape.

Using `TOPEN` Routines

With the nonstandard tape I/O package (see `TOPEN` (3F)) you can transfer blocks between the tape drive and buffers declared as Fortran 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 I/O and even has its own set of tape logical units. Refer to the man pages for complete information.

Fortran Formatted I/O for Tape

The Fortran I/O statements provide facilities for transparent access to *formatted*, sequential files on magnetic tape. (With `f77`, the tape block size can be optionally controlled by the `OPEN` statement `FILEOPT` parameter.) There is no limit on formatted record size, and records may span tape blocks.

Fortran Unformatted I/O for Tape

Using the Fortran I/O statements to connect a magnetic tape for *unformatted* access is less satisfactory. The implementation of unformatted records implies that the size of a record (+ eight 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, writing short blocks when necessary. This representation of unformatted records is preserved (even though it is inappropriate for tapes) so that 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 data file is represented on tape by a sequence of data records followed by an `endfile` record. The data is grouped into blocks, with maximum block size determined when the file is opened. The records are represented in the same way 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 records and tape blocks; that is, records can span blocks, which can contain parts of several records.

The only exception is that Fortran 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 end–of–file record in Fortran maps directly into a tape mark. In this respect, Fortran files are the same as tape system files. But since the representation of Fortran files on tape is the same as that used in the rest of UNIX, naive Fortran programs cannot read 80-column card images on tape. If you have an existing Fortran 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

As an alternative to `dd`, you can call 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 end–of–file record is encountered during execution of a `READ` statement. The standard states that the file is positioned after the end–of–file record. In real life, this means that the tape

read head is poised at the beginning of the next file on the tape. Although it seems as if you could read the next file on the tape, this is not strictly true, and is not covered by the ANSI FORTRAN 77 Language 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 end–of–file record and further manipulate the file, such as writing more records at the end, rewind the file, and reread or rewrite it.

Using Multifile Tapes

The name used to open the tape file determines certain characteristics of the connection, such as the recording density and whether the tape is automatically rewound when opened and closed.

To access a file on a tape with multiple files, first use the `mt`(1) utility to position the tape to the needed file. Then open the file as a no-rewind magnetic tape such as `/dev/nrmt0`. Referencing the tape with this name prevents it from being repositioned when it is closed. By reading the file until end-of-file and then reopening it, a program can access the next file on the tape. Any program subsequently reference the same tape can access it where it was last left, preferably at the beginning of a file, or past the end–of–file record.

However, if your program terminates prematurely, it may leave the tape positioned anywhere.

Fortran 90 I/O Considerations

Fortran 90 1.2 and Fortran 77 4.2 use different I/O libraries. However, this should be transparent to the user. Executables containing intermixed `f77` and `f90` compilations can do I/O to the same unit from both the `f77` and `f90` parts of the program.

This I/O compatibility requires that `f77` 4.2 programs be linked with `f90` 1.2 programs.

2

Program Development 3

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

Facilitating Program Builds With the `make` *Utility*

The `make` utility applies intelligence to the task of program compilation and linking. Typically, a large application may exist as a set of source files and `INCLUDE` files, which require linking with a number of libraries. Modifying any one or more of the source files requires recompilation of that part of the program and relinking. You can automate this process by specifying the interdependencies between files that make up the application along with the commands needed to recompile and relink each piece. With these specifications in a file of directives, `make` insures that only the files that need recompiling are recompiled and that relinking to build the executable uses the options and libraries you want. The following discussion provides a simple example of how to use `make`. For a summary, see `make`(1).

The `makefile`

A file called `makefile` tells `make` in a structured manner which source and object files depend on other files, and defines the commands required to compile and link them.

For example, suppose you have a program of four source files and the `makefile`:

```
demo% ls
makefile
commonblock
computepts.f
pattern.f
startupcore.f
demo%
```

Assume both `pattern.f` and `computepts.f` have an `INCLUDE` of `commonblock`, and you wish to compile each `.f` file and link the three relocatable files, along with a series of libraries, into a program called `pattern`.

The `makefile` looks like this:

```
demo% cat makefile
pattern: pattern.o computepts.o startupcore.o
    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` indicates that making `pattern` depends on `pattern.o`, `computepts.o`, and `startupcore.o`. The next line and its continuations give the command for making `pattern` from the relocatable `.o` files and libraries.

Each entry in `makefile` is a rule expressing a target object's dependencies and the commands needed to make that object. The structure of a rule is:

target`:` *dependencies-list*
`TAB` *build-commands*

- **Dependencies**—Each entry starts with a line that names the target file, followed by all the files the target depends on.
- **Commands**—Each entry has one or more subsequent lines that specify the Bourne shell commands that will build the target file for this entry. Each of these command lines must be indented by a tab character.

`make` *Command*

The `make` command can be invoked with no arguments, simply:

```
demo% make
```

The `make` utility looks for a file named `makefile` or `Makefile` in the current directory and takes its instructions from that file.

The `make` utility:

- Reads `makefile` to determine all the target files it must process, the files they depend on, and the commands needed to build them
- Finds the date and time each file was last changed
- If any target file is older than any of the files it depends on, `make` rebuilds that target, using the commands from `makefile` for that target

Macros

The `make` utility's *macro* facility allows simple parameterless string substitutions. For example, the list of relocatable files that make up the target program `pattern` can be expressed as a single macro string, making it easier to change.

A macro string definition has the form:

NAME = *string*

Use of a macro string is indicated by

$(*NAME*)

which is replaced by `make` with the actual value of the macro string named.

This example adds a macro definition naming all the object files to the beginning of `makefile`:

```
OBJ = pattern.o computepts.o startupcore.o
```

Now the macro can be used in both the list of dependencies as well as on the `f77` link command for target `pattern` in `makefile`:

```
pattern: $(OBJ)
    f77 $(OBJ) -lcore77 -lcore -lsunwindow \
    -lpixrect -o pattern
```

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

Overriding of Macro Values

The initial values of `make` macros can be overridden with command-line options to `make`. For example, with the following line to the top of `makefile`:

```
FFLAGS=-u
```

and the compile-line of `computepts.f`:

```
f77 $(FFLAGS) -c computepts.f
```

and the final link:

```
f77 $(FFLAGS) $(OBJ) -lcore77 -lcore -lsunwindow \
   lpixrect -o pattern
```

Now a simple `make` command without arguments uses the value of `FFLAGS` set above. However, this can be overridden from the command line:

```
demo% make "FFLAGS=-u -O"
```

Here, the definition of the FFLAGS macro on the make command line overrides the makefile initialization, and both the -O flag and the -u flag are passed to f77. Note that "FFLAGS=" can also be used on the command to reset the macro so that it has no effect.

Suffix Rules in make

To make writing a makefile easier, make has its own default rules that it will use depending on the suffix of a target file. Recognizing the .f suffix, make uses the f77 compiler passing as arguments any flags specified by the FFLAGS macro, the -c flag, and the name of the source file to be compiled.

The example below demonstrates this rule twice:

```
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) -c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

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

Similarly, suffix rules for .f90 files also exist to invoke the f90 compiler

More Information

A number of good, commercially published books on using make as a program development tool are currently available, including *Managing Projects with* make, by Oram and Talbott, from O'Reilly & Associates.

3

Version Tracking and Control With SCCS

SCCS stands for *Source Code Control System*. SCCS provides a way to:

- Keep track of the evolution of a source file—its change history
- Prevent a source file from being simultaneously changed by other developers
- Keep track of the version number by providing version stamps

The basic three operations of SCCS are:

- Putting files under SCCS control
- Checking out a file for editing
- Checking in a file

This section shows you how to use SCCS to perform these tasks, using the previous program as an example. Only basic SCCS is described and only three SCCS commands are introduced: `create`, `edit`, and `delget`.

Controlling Files With SCCS

Putting files under SCCS control involves:

- Making the SCCS directory
- Inserting SCCS ID keywords into the files (this is optional)
- Creating the SCCS files

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 of using keywords is that the version information appears in the source listing and compiled object program. If preceded by the string `@(#)`, the keywords in the object file can be printed using the `what` command.

Included header files that contain only parameter and data definition statements do not generate any initialized data, so the keywords for those files usually are put in comments or in parameter statements. Some files, like ASCII data files or `makefile`s, the SCCS information appears in comments.

SCCS keywords appear in the form `%`*keyword*`%` and are expanded into their values by the SCCS get command. The most commonly used keywords are:

`%Z%` expands to the identifier string `@(#)` recognized by the `what` command.

`%M%` expands to the name of the source file.

`%I%` expands to the version number of this SCCS maintained file.

`%E%` expands to the current date.

For example, we could identify the makefile with a `make` comment containing these keywords:

```
#   %Z%%M% %I% %E%
```

The source files, `startupcore.f`, `computepts.f`, and `pattern.f` can be identified by initialized data of the form:

```
    CHARACTER*50 SCCSID
    DATA SCCSID/"%Z%%M% %I% %E%\n"/
```

When this file is processed by SCCS and then compiled and the object file processed by the what command, the following will be displayed:

```
demo% f77 -c pattern.f
...
demo% what pattern
pattern:
    pattern.f 1.2 96/06/10
```

You can also create a PARAMETER named CTIME that is automatically updated whenever the file is accessed with get.

```
    CHARACTER*(*) CTIME
    PARAMETER ( CTIME="%E%")
```

INCLUDE files can be annotated with a Fortran comment containing the SCCS stamp:

```
C   %Z%%M% %I% %E%
```

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%
```

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 modified 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.

More Information

We recommend the book *Applying RCS and SCCS*, by Bolinger and Bronson, from O'Reilly & Associates.

3

Libraries 4

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

Understanding Libraries

A software *library* is usually a set of subprograms that have been previously compiled and organized into a single binary *library file*. Each member of the set is called a library *element* or *module*. The linker searches the library files, loading object modules referenced by the user program while building the executable binary program. See `ld`(1) and the Sun *Linker and Libraries Guide* for details.

There are two basic kinds of software libraries:

- **Static library**—A library in which modules are bound into the executable file *before* execution. Static libraries are commonly named `lib`*name*`.a`. The `.a` suffix refers to *archive*.
- **Dynamic library**—A library in which modules can be bound into the executable program at runtime. Dynamic libraries are commonly named `lib`*name*`.so`. The `.so` suffix refers to *shared object*.

Typical system libraries that have both static and dynamic versions are:

- Fortran libraries: `libF77.a and libF77.so`
- VMS Fortran libraries: `libV77.a and libV77.so`
- C libraries: `libc.a and libc.so`

Advantages of Libraries

Library files provide an easy way for programs to share commonly used subroutines. You need only name the library when linking the program, and those library modules that resolve references in the program are linked and merged into the executable file.

There are two advantages to the use of libraries:

- There is no need to have source code for the library routines that a program calls.
- Only the needed modules are loaded.

Linker Debugging Options

Summary information about library usage and loading can be obtained by passing additional options to the linker on the compile command line, either by using the option syntax `-Qoption ld` *linker_option* or by setting the environment variable `LD_OPTIONS`.

Using `LD_OPTIONS` environment variable:

```
demo% setenv LD_OPTIONS "-m -Dfiles"
demo% f77 -o myprog myprog.f
```

is equivalent to:

```
demo% f77 -o myprog -Qoption ld -m -Qoption ld -Dfiles myprog.f
```

Some linker options have their compiler command-line equivalents and can appear directly on the `f77` or `f90` command: `-B`*x*, `-d`*x*, `-G`, `-h`*name*, `-R`*path*, and `-ztext`.

More detailed examples and explanations of linker options and environment variables can be found in the Solaris *Linker and Libraries Guide*.

Generating a Load Map

The linker `-m` option generates a load map that displays library linking information listing the routines linked during the building of the executable binary program. Routines are listed together with the libraries that they come from.

Example: `-m` for load map:

```
demo% f77 -Qoption ld -m any.f
any.f:
 MAIN:
       LINK EDITOR MEMORY MAP


output  input      virtual
section section    address    size

.interp            100d4      11
        .interp    100d4      11 (null)
.hash              100e8     2e8
        .hash      100e8     2e8 (null)
.dynsym            103d0     650
        .dynsym    103d0     650 (null)
.dynstr            10a20     366
        .dynstr    10a20     366 (null)
.text              10c90    1e70
        .text      10c90      00 /set/lang/sparc-S2/SC4.2/lib/crti.o
        .text      10c90      f4 /set/lang/sparc-S2/SC4.2/lib/crt1.o
        .text      10d84      00 /set/lang/sparc-S2/SC4.2/lib/values-xi.o
        .text      10d88     d20 sparse.o
...etc
```

Listing Other Information

Solaris 2.3 and later has additional linker debugging features, available through the linker's `-D`*keyword* option. A complete list can be displayed using `-Dhelp`.

Example: List linker debugging aid options using `-Dhelp` option:

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

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

```
demo% f77 -Qoption ld -Dfiles direct.f
direct.f:
 MAIN direct:
debug: file=/opt/SUNWspro/SC4.2/lib/crti.o  [ ET_REL ]
debug: file=/opt/SUNWspro/SC4.2/lib/crt1.o  [ ET_REL ]
debug: file=/opt/SUNWspro/SC4.2/lib/values-xi.o  [ ET_REL ]
debug: file=direct.o  [ ET_REL ]
debug: file=/opt/SUNWspro/SC4.2/lib/libM77.a  [ archive ]
debug: file=/opt/SUNWspro/lib/libF77.so  [ ET_DYN ]
debug: file=/opt/SUNWspro/SC4.2/lib/libsunmath.a  [ archive ]
    …
```

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

Consistent Compiling and Linking

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

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

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

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

Library Search Paths and Order

The linker searches for libraries at several locations and in a certain prescribed order. Some of these locations are standard paths, while others depend on the compiler options `-R`*path*, `-l`*library* and `-L`*dir* and the environment variable `LD_LIBRARY_PATH`.

Search Order for Standard Library Paths

The standard library search paths used by the linker are determined by the installation path, and they differ for static and dynamic loading.

The base directory, here called *BaseDir*, is defined as follows:

	Standard Install	**Nonstandard Install to** `/my/dir/`
BaseDir =	`/opt/SUNWspro/`	`/my/dir/SUNWspro/`

Static Linking

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

- *BaseDir*`/lib` Sun shared libraries
- `/usr/ccs/lib/` Standard location for SVr4 software
- `/usr/lib` Standard location for UNIX software

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

Dynamic Linking

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

- Paths specified by user with `-R`*path*
- `/`*BaseDir*`/lib/`
- `/usr/lib` standard UNIX default

The search paths are built into the executable.

Library Search Path and Order — Static Linking

Use the `-l`*library* compiler option to name additional libraries for the linker to search when resolving external references. For example, the option `-lmylib` adds the library `libmylib.so` or `libmylib.a` to the search list.

The linker looks in the standard directory paths to find the additional `libmylib` library. The `-L` option (and the `LD_LIBRARY_PATH` environment variable) creates a list of paths that tell the linker where to look for libraries outside the standard paths.

Were `libmylib.a` in directory `/home/proj/libs`, then the option `-L/home/proj/libs` would tell the linker where to look when building the executable:

```
demo% f77 -o pgram part1.o part2.o -L/home/proj/libs -lmylib
```

Command-Line Order for `-l`*library* Options

For any particular unresolved reference, libraries are searched only once, and only for symbols that are undefined at that point in the search. If you list more than one library on the command line, then the libraries are searched in the order they are found on the command line. Place `-l`*library* options as follows:

- Place the `-l`*library* option after any `.f`, `.for`, `.F`, `.f90`, or `.o` files.
- If you call functions in `lib`*x*, and they reference functions in `lib`*y*, then place `-l`*x* before `-l`*y*.

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

The `-L`*dir* option adds the *dir* directory path to the library search list. The linker searches for libraries first in any directories specified by the `-L` options and then in the standard directories. This option is useful only if it is placed *preceding* the `-l`*library* options to which it applies.

LD_LIBRARY_PATH *Environment Variable*

Use the environment variable LD_LIBRARY_PATH to specify directory paths the linker should search for libraries specified with the -l*library* option. Using this environment variable would make the previous example look like:

```
demo% setenv LD_LIBRARY_PATH /home/proj/libs
demo% f77 -o pgram part1.o part2.o -lmylib
```

Multiple directories can be specified, separated by a colon. In the most general case, the LD_LIBRARY_PATH variable may contain two lists of colon-separated directories separated by a semicolon:

dirlist1;*dirlist2*

The directories in *dirlist1* are searched first, followed by any explicit -L*dir* directories specified on the command line, followed then by *dirlist2* and the standard directories.

That is, if the compiler is called with any number of occurrences of -L, as in:

f77 ... -L*path1* ... -L*pathn* ...

then the search ordering is:

dirlist1 path1 ... pathn dirlist2 standard_paths

When the LD_LIBRARY_PATH variable contains only a single colon-separated list of directories, it is interpreted as *dirlist2*.

Note – Use of this environment variable with production software is strongly discouraged. Although useful as a temporary mechanism for influencing the runtime linker's search path, *any* dynamic executable that can reference this environment variable will have its search paths altered, which could cause unexpected results or a degradation in performance.

Library Search Path and Order — Dynamic Linking

Changing the library search path and order of loading with dynamic libraries differs from the static case in that actual linking takes place at runtime rather than build time.

Specifying Dynamic Libraries at Build Time

When *building* the executable file, the linker records the paths to shared libraries into the executable itself. These search paths can be specified by using the `-R`*path* option. (Contrast with the `-L`*dir* option which indicates where at buildtime to find the library specified by a `-l`*library* option, but does not record this path into the binary executable.)

The directory paths that were built in when the executable was created can be viewed using the `dump` command.

Example: List the directory paths built into `a.out`:

```
demo% f77 program.f -R/home/proj/libs -L/home/proj/libs -lmylib
demo% dump -Lv a.out | grep RPATH
[5] RPATH    /home/proj/libs:/opt/SUNWspro/lib
```

Specifying Dynamic Libraries at Runtime

At *runtime*, the linker determines where to find the dynamic libraries an executable needs from:

- the value of `LD_LIBRARY_PATH` at runtime
- the paths that had been specified by `-R` at the time the executable file was built.

As noted earlier, use of `LD_LIBRARY_PATH` can have unexpected side-effects and is not recommended.

Errors During Dynamic Linking

When the dynamic linker cannot locate a needed library, it issues the error message:

```
ld.so: prog: fatal: libmylib.so: can't open file:
```

The possible causes might be:

- The libraries are not where they are supposed to be.

Perhaps you specified paths to shared libraries when the executable was built, but the libraries have subsequently been moved. For example, you built `a.out` with your own dynamic libraries in */my/libs/*, and then sometime later moved the libraries to another directory.

Use `ldd` to determine where the executable expects to find the libraries:

```
demo% ldd a.out
    libsolib.so => /export/home/proj/libsolib.so
    libF77.so.3 => /opt/SUNWspro/lib/libF77.so.3
    libc.so.1 => /usr/lib/libc.so.1
    libdl.so.1 => /usr/lib/libdl.so.1
```

If possible, move or copy the libraries into the proper directory or make a soft link to the directory (using `ln -s`) in the directory that the linker is searching.

- `LD_LIBRARY_PATH` is not set correctly.

  Check that `LD_LIBRARY_PATH` at runtime includes the path to the needed libraries.

Creating Static Libraries

Static library files are built from precompiled object files (`.o` files) using the `ar`(1) utility.

The linker extracts from the library any elements whose entry points are referenced within the program it is linking, such as a subprogram, entry name, or `COMMON` block initialized in a `BLOCKDATA` subprogram. These extracted elements (routines) are bound permanently into the `a.out` executable file generated by the linker.

Tradeoffs

There are three main issues to keep in mind regarding static, as compared to dynamic, libraries and linking:

- Static libraries are more self contained but less adaptable.

  If you bind an `a.out` executable file *statically*, the library routines it needs become part of the executable binary. However, if it becomes necessary to update a static library routine bound into the `a.out` executable, the entire

a.out file must be relinked and regenerated to take advantage of the updated library. With *dynamic* libraries, the library is not part of the a.out file and linking is done at runtime. To take advantage of an updated dynamic library, all that is required is that the new library be installed on the system.

- The "elements" in a static library are individual compilation units, .o files.

  Since a single compilation unit (a source file) can contain more than one subprogram, these routines when compiled together become a single module in the static library. This means that *all* the routines in the compilation unit are loaded together into the a.out executable, even though only one of those subprograms was actually called. This situation can be improved by optimizing the way library routines are distributed into compilable source files. (Still, only those library modules actually referenced by the program are loaded into the executable.)

- Order matters when linking static libraries.

  The linker processes its input files in the order in which they appear on the command line—left to right. When the linker decides whether or not to load an element from a library, its decision is determined by the library elements that it has already processed. This order is not only dependent on the order of the elements as they appear in the library file but also on the order which the libraries are specified on the compile command line.

  Example: If the Fortran program is in two files, main.f and crunch.f, and only the latter accesses the Sun Performance Library library, it is an error to reference that library before crunch.f or crunch.o:

```
demo% f77 main.f -lsunperf crunch.f -o myprog     (Incorrect)
demo% f77 main.f crunch.f -lsunperf -o myprog     (Correct)
```

Creation of a Simple Static Library

Suppose that we can distribute all the routines in a program over a group of source files and that these files are wholly contained in the subdirectory test_lib/.

Suppose further that the files are organized in such a way that they each contain a single principal subprogram that would be called by the user program, along with any "helper" routines that the subprogram may call but

which are called from no other routine in the library. Also, any helper routines called from more than one library routine are gathered together into a single source file. This gives a reasonably well-organized set of source and object files.

Assume that the name of each source file is taken from the name of the first routine in the file, which in most cases is one of the principal files in the library:

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

The lower-level "helper" routines are gathered together into the file **etc.f**. The other files may contain one or more subprograms.

First, compile each of the library source files, using the `-c` option, to generate the corresponding relocatable `.o` files:

```
demo% f77 -c *.f
delte.f:
    delte:
    q_fixx:
dropx.f:
    dropx:
etc.f:
    q_fill:
    q_step:
    q_node:
    q_warn:
 ...etc
demo% ls
total 42
 2 dropx.f     4 etc.o      2 linkz.f    4 markx.o
 2 delte.f     4 dropx.o     2 evalx.f    4 linkz.o    2 point.f
 4 delte.o     2 etc.f      4 evalx.o    2 markx.f    4 point.o
demo%
```

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

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

To use this library, either include the library file on the compilation command or use the `-l` and `-L` compilation options.

Using the `.a` file directly:

```
demo% cat trylib.f
C    program to test testlib routines
    x=21.998
    call evalx(x)
    call point(x)
    print*, 'value ',x
    end
demo% f77 -o trylib trylib.f test_lib/testlib.a
trylib.f:
 MAIN:
demo%
```

Notice that the main program only calls two of the routines in the library. You can verify that the uncalled routines in the library were not loaded into the executable file by looking for them in the list of names in the executable displayed by `nm`:

```
demo% nm trylib | grep FUNC | grep point
[146]|     70016|     152|FUNC |GLOB |0    |8      |point_
demo% nm trylib | grep FUNC | grep evalx
[165]|     69848|     152|FUNC |GLOB |0    |8      |evalx_
demo% nm trylib | grep FUNC | grep delte
demo% nm trylib | grep FUNC | grep markx
demo% ..etc
```

In the preceding example, `grep` finds entries in the list of names only for those library routines that were actually called.

Another way to reference the library is through the `-l`*library* and `-L`*path* options. Here, the library's name would have to be changed to conform to the `lib`*name*`.a` convention:

```
demo% mv test_lib/testlib.a test_lib/libtestlib.a
demo% f77 -o trylib trylib.f -Ltest_lib -ltestlib
trylib.f:
 MAIN:
```

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

```
demo% f77 -o myprog myprog.f -L/usr/local/lib -ltestlib
```

Replacement in a Static Library

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

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

```
demo% f77 -c point.f
demo% ar r testlib.a point.o
demo%
```

Ordering Routines in a Static Library

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

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

4

Creating Dynamic Libraries

Dynamic library files are built by the linker `ld` from precompiled object modules that can be bound into the executable file *after* execution begins.

Another feature of a dynamic library is that modules can be used by other executing programs in the system *without* duplicating modules in each program's memory. For this reason, a dynamic library is also a *shared* library.

A dynamic library offers the following features:

- The object modules are *not* bound into the executable file by the linker during the compile-link sequence; such binding is deferred until runtime.
- A shared library module is bound into system memory when the first running program references it. If any subsequent running program references it, that reference is mapped to this first copy.
- Maintaining programs is easier with dynamic libraries. Installing an updated dynamic library on a system immediately affects all the applications that use it without requiring relinking of the executable.

Tradeoffs

Dynamic libraries introduce some additional tradeoff considerations:

- **Smaller `a.out` file**

  Deferring binding of the library routines until execution time means that the size of the executable file is less than the equivalent executable calling a static version of the library; the executable file does not contain the binaries for the library routines.

- **Possibly smaller process memory utilization**

  When several processes using the library are active simultaneously, only one copy of the memory resides in memory and is shared by all processes.

- **Possibly increased overhead**

  Additional processor time is needed to load and link-edit the library routines during runtime. Also, the library's position-independent coding may execute more slowly than the relocatable coding in a static library.

- **Possible overall system performance improvement**

Reduced memory utilization due to library sharing should result in better overall system performance (reduced I/O access time from memory swapping).

Performance profiles among programs vary greatly from one to another. It is not always possible to determine or estimate in advance the performance improvement (or degradation) between dynamic versus static libraries. However, if both forms of a needed library are available to you, it would be worthwhile to evaluate the performance of your program with each.

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. Such code is inherently sharable between simultaneous processes. Thus, if you are building a dynamic, shared library, you must compile the component routines to be position-independent (by using compiler options `-pic` or `-PIC`).

In position-independent code, each reference to a global item is compiled as a reference through a pointer into a global offset table. Each function call is compiled in a relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8Kbytes on SPARC processors. The `-PIC` compiler option is similar to `-pic`, but `-PIC` allows the global offset table to span the range of 32–bit addresses.

Binding Options

You can specify dynamic or static library binding when you compile. These options are actually linker options, but they are recognized by the compiler and passed on to the linker.

`-Bdynamic | -Bstatic`

`-Bdynamic` sets the preference for shared, dynamic binding whenever possible. `-Bstatic` restricts binding to static libraries only.

When both static and dynamic versions of a library are available, use this option to toggle between preferences on the command line:

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

-dy | -dn

Allows or disallows dynamic linking for the entire executable. (This option may appear only once on the command line.)

-dy allows dynamic, shared libraries to be linked. -dn does not allow linking dynamic libraries.

Naming Conventions

To conform to the dynamic library naming conventions assumed by the link loader and the compilers, assign names to the dynamic libraries that you create with the prefix lib and the suffix .so. For example, libmyfavs.so could then be referenced by the compiler option -lmyfavs.

The linker also accepts an optional version number suffix: for example, libmyfavs.so.1 for version *one* of the library, etc.

The compiler's -h*name* option records *name* as the name of the dynamic library being built.

A Simple Dynamic Library

Building a dynamic library requires a compilation of the source files with the -pic or -PIC option and linker options -G, -ztext, and -h*name*. These linker options are available through the compiler command line.

You can create a dynamic library with the same files used in the static library example.

Example: compile with `-pic` and other linker options:

```
demo% f77 -o libtestlib.so.1 -G -pic -ztext -hlibtestlib.so.1 *.f
delte.f:
    delte:
    q_fixx:
dropx.f:
    dropx:
etc.f:
    q_fill:
    q_step:
    q_node:
    q_warn:
evalx.f:
    evalx:
linkz.f:
    linkz:
markx.f:
    markx:
point.f:
    point:
Linking:
```

`-G` tells the linker to build a dynamic library.

`-ztext` warns you if it finds anything other than position-independent code, such as relocatable text.

Example: Bind—make an executable file `a.out` using the dynamic library:

```
demo% f77 -o trylib -R‘pwd‘ trylib.f libtestlib.so.1
trylib.f:
 MAIN main:
demo% file trylib
trylib: ELF 32-bit MSB executable SPARC Version 1, dynamically
linked, not stripped
demo% ldd trylib
    libtestlib.so.1 => /export/home/U/Tests/libtestlib.so.1
    libF77.so.3 => /opt/SUNWspro/lib/libF77.so.3
    libc.so.1 => /usr/lib/libc.so.1
    libdl.so.1 => /usr/lib/libdl.so.1
```

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

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

The `ldd` command shows that the executable, `trylib`, uses some shared libraries, including our `libtestlib.so.1`; `libf77`, `libdl`, and `libc` are included by default by `f77`. It also shows exactly which files on the system are used for these libraries.

Libraries Provided with Sun Fortran Compilers

Table 4-1 shows the libraries are installed with the compilers:

Table 4-1 Major Libraries Provided With the Compilers

Library	Name	Options Needed
`f77` functions, nonmath	`libF77`	None
`f77` functions, nonmath, multithread safe	`libF77_mt`	`-parallel`
`f77` math library	`libM77`	None
VMS library	`libV77`	`-lV77`
Library used with Pascal, Fortran, and C	`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`

See also the `math_libraries` README file for more information.

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 Library Reference* and the *Fortran 77 Language Reference* for details on these routines.

`POSIX` *Library*

There are two versions of `POSIX` bindings provided with the compilers:

- `libFposix`, which is just the bindings (`-lFposix`)
- `libFposix_c`, which does some runtime checking to make sure you are passing correct handles (`-lFposix_c`)

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.

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.

4

Shippable Libraries

If your executable uses a Sun dynamic library that is listed in the following README 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.

Program Analysis and Debugging 5

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

Global Program Checking (f77 Only)

The f77 compiler's –Xlist*x* options provide a valuable way to analyze a source program for inconsistencies and possible runtime problems. The analysis performed by the compiler is *global*, across subprograms. –Xlist*x* reports errors in alignment, agreement in number and type for subprogram arguments, common block, parameter, and various other kinds of errors.

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

Note – Although a subset of -Xlist options are available with this release (1.2) of f90, a conventional cross-reference map is produced but global program checking is not performed; full checking will appear in a subsequent release of the f90 compiler.

GPC Overview

Global program checking (GPC), invoked by the `-Xlist`*x* option, does the following:

- Enforces type-checking rules of Fortran more stringently than usual, especially between separately compiled routines
- Enforces some portability restrictions needed to move programs between different machines or operating systems
- Detects legal constructions that nevertheless may be suboptimal or error-prone
- Reveals other potential bugs and obscurities

In particular, global-cross checking reports problems such as:

- Interface problems
  - Conflicts in number and type of dummy and actual arguments
  - Wrong types of function values
  - Possible conflicts due to data type mismatches in common blocks between different subprograms
- Usage problems
  - Function used as a subroutine or subroutine used as a function
  - Declared but unused functions, subroutines, variables, and labels
  - Referenced but not declared functions, subroutines, variables, and labels
  - Usage of unset variables
  - Unreachable statements
  - Implicit type variables
  - Inconsistency of the named common block lengths, names, and layouts
- Syntax problems – syntax errors found in a Fortran program
- Portability problems – code that does not conform to ANSI Fortran, if the appropriate option is used

How to Invoke Global Program Checking

The `-Xlist` option on the command line invokes the compiler's global program analyzer. There are a number of `–Xlist`*x* suboptions, as described in the sections that follow.

Example: Compile three files for basic global program checking:

```
demo% f77 –Xlist  any1.f  any2.f  any3.f
```

In the preceding example, the compiler:

- Produces output listings in the file `any1.lst`
- Compiles and links the program if there are no errors

Screen Output

Normally, output listings produced by `–Xlist`*x* are written to a file. To display directly to the screen, use `–Xlisto` to write the output file to `/dev/tty`.

Example: Display to terminal:

```
demo% f77 –Xlisto /dev/tty  any1.f
```

Default Output Features

The `–Xlist` option provides a combination of features available for output. With no other `-Xlist` options, you get the following by default:

- The listing file name is taken from the first input source or object file that appears, with the extension replaced by `.lst`
- A line-numbered source listing
- Error messages (embedded in listing) for inconsistencies across routines
- Cross-reference table of the identifiers
- Pagination at 66 lines per page and 79 columns per line
- No call graph
- No expansion of `include` files

File Types

The checking process recognizes all the files in the compiler command line that end in `.f`, `.f90`, `.for`, `.F`, `.o`, or `.s`. The `.o` and `.s` files supply the process with information regarding global names only, such as subroutine and function names.

Analysis Files (`.fln` Files)

The compiler saves individual source file analysis results into files with a `.fln` suffix. It usually uses the source directory. You can specify an alternate directory to receive these files with the `-Xlistfln`*dir* option.

```
demo% f77 -Xlistfln/tmp *.f
```

Libraries compiled with `-Xlist` options have their analysis files built into the binary files automatically, enabling GPC over programs that link these libraries.

Some Examples of `-Xlist` *and Global Program Checking*

Source code used in the following examples, `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
```

Example: Use `-XlistE` to show errors and warnings:

```
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%
```

Compiling the same program with `-Xlist` also produces a cross-reference table:

Output File: `f77 -Xlist Repeat.f`

```
         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*)

```
 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 preceding 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; used at lines 9 and 10
  - In `prnok`, at line 20, defined; at line 21, used as an argument

Suboptions for Global Checking Across Routines

The basic global cross-checking option is `-Xlist` with no suboption. It is a combination of suboptions, each of which could have been specified separately.

Described below are options for producing the listing, errors, and cross-reference table. Multiple suboptions may appear on 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 specials

Example: Each of these two command lines performs 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:

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

Here is a summary of all `-Xlist` suboptions:

Option	Action
`-Xlist` (*no suboption)*	Show errors, listing, and cross-reference table
`-Xlistc`	Show call graphs and errors
`-XlistE`	Show errors
`-Xlisterr[`*nnn*`]`	Suppress error *nnn* in the verification report
`-Xlistf`	Produce fast output
`-Xlistfln`*dir*	Put the `.fln` files in *dir*
`-Xlisth`	Errors from cross-checking stop compilation
`-XlistI`	List and cross-check `include` files
`-XlistL`	Show the listing and errors
`-Xlistl`*n*	Set page breaks
`-Xlisto` *name*	Rename the `-Xlist` output report file
`-Xlists`	Unreferenced symbols suppressed from the cross-reference
`-Xlistv`*n*	Show different amounts of semantic information
`-Xlistw[`*nnn*`]`	Set the width of output lines
`-Xlistwar[`*nnn]*	Suppress warning *nnn* in the report
`-XlistX`	Show the cross-reference table and errors

–Xlist *Suboptions Reference*

–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 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 without this option is to generate object files.

–Xlistfln*dir* Put the .fln files into the *dir* directory, which must already exist.

The default is the source directory.

–Xlisth 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.

–XlistI List and cross-check `include` files.

If `–XlistI` is the only suboption used, `include` files are shown or scanned along with the standard `–Xlist` output (line numbered listing, error messages, and a cross-reference table).

- **Listing**—If the listing is not suppressed, then the `include` files are listed in place. Files are listed as often as they are included. The files are:
  - Source files
  - `#include` files
  - `INCLUDE` files
- **Cross-Reference 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 not to show `include` files.

–XlistL 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.

–Xlistl*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.

With *n*=0 (`-Xlistl0`) this option shows listings and cross-references with no page breaks for easier on-screen viewing.

–Xlisto *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 screen, use the command: `–Xlisto /dev/tty`

–Xlists 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, line numbers, and 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, line numbers, and 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 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 answers the following questions about each identifier:

- Is it an argument?
- Does it appear in a `COMMON` or `EQUIVALENCE` declaration?
- Is it set or used?

Some Examples Using Suboptions

Example: Use –Xlistwar*nnn* to suppress two warnings from a preceding example:

```
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%
```

Example: Explain a message and find a type mismatch:

ShoGetc.f

```
demo% cat ShoGetc.f
    CHARACTER*1 c
    i = getc(c)
    END
demo% f77 -silent ShoGetc.f
demo% a.out
Z
```

Program waits for input
Type Z on keyboard →

The problem:
Why this message?

```
 Note: IEEE floating-point exception flags raised:
    Invalid Operation;
 See the Numerical Computation Guide, ieee_flags(3M)
```

Compile with -Xlist
List the output.

```
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
```

Here is the error.

The default typing of getc is not consistent with the Fortran library.
f77 has information about the Fortran library — in particular, that getc is integer.

The solution: →
Declare getc an integer.

```
demo% cat ShoGetc.f
    CHARACTER*1 c
    INTEGER getc
    i = getc(c)
    END
demo% f77 -silent ShoGetc.f
demo% a.out
Z
demo%
```

No error message.

Special Compiler Options

Some compiler options are useful for debugging. They check subscripts, spot undeclared variables, show stages of the compile-link sequence, display versions of software, and so on.

With Solaris 2.3 and later, there are new linker debugging aids. See `ld`(1), or run `ld -Dhelp` to see online documentation.

Subscript Bounds (`-C`)

The `-C` option adds checks for out-of-bounds array subscripts.

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

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`)

The `-u` option checks for any undeclared variables.

The `-u` option causes all variables to be initially identified as undeclared, so that all variables that are not explicitly declared are flagged with an error. The `-u` flag is useful for discovering mistyped variables. If `-u` is set, all variables are treated as undeclared until explicitly declared. Use of an undeclared variable is accompanied by an error message.

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.

Interactive Debugging With dbx *and The WorkShop*

The Sun WorkShop provides a tightly integrated development environment for building and browsing, as well as debugging applications written in Fortran, C, C++, and Pascal.

The WorkShop debugging facility is a window-based interface to dbx, while dbx itself is an interactive, line-oriented, source-level symbolic debugger. Either can be used to determine where a program crashed, to view or trace the values of variables and expressions in a running code, and to set breakpoints.

The WorkShop adds a sophisticated graphical environment to the debugging process that is integrated with tools for editing, building, and source code version control. It includes a data visualization capability to display and explore large and complex datasets, simulate results, and interactively steer computations.

For details, see the Sun manuals *WorkShop: Getting Started* and *WorkShop: Command-Line Utilities*, and the dbx(1) man pages.

The dbx program provides event management, process control, and data inspection. You can watch what is happening during program execution, and perform the following tasks:

- Fix one routine, then continue executing without recompiling the others
- Set watchpoints to stop or trace if a specified item changes
- Collect data for performance tuning
- 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

- 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
- Follow programs as they `fork`(2) and `exec`(2)

Debugging Optimized Programs

To debug optimized programs, use the `dbx fix` command to recompile the routines you want to debug:

- Compile the program with the appropriate `–O`*n* optimization level.
- Start the execution under `dbx`.
- Use `fix –g` *any*`.f` without optimization on the routine you want to debug.
- Use `continue` with that routine compiled.

Some optimizations may be inhibited by the presence of `-g` on the compilation command. For example, `-g` suppresses the automatic inlining usually obtained with `-O4`. `-g` cancels any parallelization option (`-autopar`, `-explicitpar`, `-parallel`), as well as `-depend` and `-reduction`. Debugging is facilitated by specifying `-g` without any optimization options. See the `dbx` documentation for details.

Viewing Compiler Listing Diagnostics

The `error` utility program can be used to view compiler diagnostics merged with the source code. `error` inserts compiler diagnostics above the relevant line in the source file. The diagnostics include the standard compiler error and warning messages, but *not* the `–Xlist` error and warning messages.

Warning – This utility rewrites your source files and does not work if the source files are read-only, or in a read–only directory.

`error`(1) is available if the operating system was installed with a developer install, rather than an end–user install; it can also be installed from the package, `SUNWbtool`.

Facilities also exist in the Sun WorkShop for viewing compiler diagnostics. Refer to the Sun *WorkShop: Getting Started* guide.

5

Floating-Point Arithmetic 6

This chapter considers floating-point arithmetic and suggests strategies for avoiding and detecting numerical computation errors.

For a detailed examination of floating-point computation on SPARC, Intel, and PowerPC processors, the Sun *Numerical Computation Guide* is strongly recommended. It includes the valuable paper "What Every Computer Scientist Should Know About Floating-point Arithmetic," by David Goldberg.

Introduction

Sun's floating-point environment on SPARC, Intel, and PowerPC implements the arithmetic model specified by the IEEE Standard 754 for Binary Floating Point Arithmetic. This environment enables you to develop robust, high-performance, portable numerical applications. It also provides tools to investigate any unusual behavior by a numerical program.

In numerical programs, there are many potential sources for computational error:

- The computational model may be wrong.
- The algorithm used may be numerically unstable.
- The data may be ill-conditioned.
- The hardware may be producing unexpected results.

6

Finding the source of the errors in a numerical computation that has gone wrong can be extremely difficult. The chance of coding errors can be reduced by using commercially available and tested library packages whenever possible. Choice of algorithms is another critical issue. Using the appropriate computer arithmetic is another.

This chapter makes no attempt to teach or explain numerical error analysis. The material presented here is intended to introduce the IEEE floating-point model as implemented by Sun's Fortran compilers.

IEEE Floating-Point Arithmetic

IEEE arithmetic is a relatively new way of dealing with arithmetic operations that result in such problems as invalid, division by zero, overflow, underflow, or inexact. The differences are in rounding, handling numbers near zero, and handling numbers near the machine maximum.

The IEEE standard supports user handling of exceptions, rounding, and precision. Consequently, the standard supports interval arithmetic and diagnosis of anomalies. IEEE Standard 754 makes it possible to standardize elementary functions like `exp` and `cos`, to create high precision arithmetic, and to couple numerical and symbolic algebraic computation.

IEEE arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic. The standard simplifies the task of writing numerically sophisticated, portable programs. Many questions about floating-point arithmetic concern elementary operations on numbers. For example:

- What is the result of an operation when the infinitely precise result is not representable in the computer hardware?
- Are elementary operations like multiplication and addition commutative?

Another class of questions concerns floating-point exceptions and exception handling. What happens if you:

- Multiply two very large numbers with the same sign?
- Divide nonzero by zero?
- Divide zero by zero?

In older arithmetic models, the first class of questions might not have the expected answers, while the exceptional cases in the second class might all have the same result: the program aborts on the spot or proceeds with garbage results.

The standard ensures that operations yield the mathematically expected results with the expected properties. It also ensures that exceptional cases yield specified results, unless the user specifically makes other choices.

For example, the exceptional values `+Inf`, `-Inf`, and `NaN` are introduced intuitively:

`big*big = +Inf`	*Positive infinity*
`big*(-big) = -Inf`	*Negative infinity*
`num/0.0 = +Inf`	*Where* `num > 0.0`
`num/0.0 = -Inf`	*Where* `num < 0.0`
`0.0/0.0 = NaN`	*Not a Number*

Also, five types of floating-point exception are identified:

- *Invalid*—Operations with mathematically invalid operands— for example, 0.0 / 0.0, sqrt(-1.0), and log(-37.8)
- *Division by zero*—Divisor is zero and dividend is a finite nonzero number— for example, 9.9 / 0.0
- *Overflow*—Operation produces a result that exceeds the range of the exponent— for example, `MAXDOUBLE`+0.0000000000001e308
- *Underflow*—Operation produces a result that is too small to be represented as a normal number— for example, `MINDOUBLE * MINDOUBLE`
- *Inexact*—Operation produces a result that cannot be represented with infinite precision— for example, 2.0 / 3.0, log(1.1) and 0.1 in input

Sun's implementation of the IEEE standard is described in the Sun *Numerical Computation Guide*.

Handling Exceptions

Exception handling according to the IEEE standard is the default on SPARC, Intel, and PowerPC processors. However, there is a difference between detecting a floating-point exception and generating a signal for a floating-point exception (`SIGFPE`).

Following the IEEE standard, two things happen when an untrapped exception occurs during a floating-point operation:

- The system returns a default result.
  For example, on 0/0 (*invalid*), return `NaN` as the result.
- A flag is set to indicate that an exception is raised.
  For example, 0/0 (*invalid*), set "invalid operation" flag.

Trapping a Floating-Point Exception–`f77` vs `f90`

With `f77`, the default on SPARC, Intel, and PowerPC systems is *not* to automatically generate a signal to interrupt the running program for a floating-point exception. The assumptions are that signals could degrade performance and that most exceptions are not significant as long as expected values are returned.

The default with `f90` is to automatically trap on division by zero, overflow, and invalid operation. Compiling an application's main program unit with the `f90` option `–fnonstop` changes this default behavior to be the same as `f77`'s. In the discussions that follow, `f77`'s default behavior, or `f90 -fnonstop`, is assumed.

To enable exception trapping, compile the main program with one of the `-ftrap` options— for example, `-ftrap=common`.

IEEE Routines

The following interfaces help people use 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 an exception handlerroutine

- `ieee_functions`(3m)—List name and purpose of each IEEE function
- `ieee_values`(3m)—List functions that return special values
- Other `libm` functions:
  - `ieee_retrospective`
  - `nonstandard_arithmetic`
  - `standard_arithmetic`

The SPARC processors conform to the IEEE standard in a combination of hardware and software support for different aspects. Intel and PowerPC processors conform to the IEEE standard entirely through hardware support.

The newest SPARC processors contain floating-point units with integer multiply and divide instructions and hardware square root.

Best performance is obtained when the compiled code properly matches the runtime floating-point hardware. The compiler's `-xtarget=` option permits specification of the runtime hardware. For example, `-xtarget=ultra` would inform the compiler to generate object code that will perform best on an UltraSPARC processor.

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

Flags and `ieee_flags()`

The `ieee_flags` function is used to query and clear exception status flags. It is part of the `libsunmath` library shipped with Sun compilers and performs the following tasks:

- 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:

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

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

Possible parameter values are shown in the following table:

```
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
```

Note that these are literal character strings, and the output parameter *out* must be at least `CHARACTER*9`. The meanings of the possible values for *in* and *out* depend on the action and mode they are used with. These are summarized in Table 6-1.

Table 6-1 `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

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

```
     CHARACTER *9, out
    ieeer = ieee_flags( 'get', 'exception', '', out )
    PRINT *, out, ' flag raised'
```

Also, to determine if the `overflow` exception flag is raised, set the input argument *in* to `overflow`. On return, if `out` equals `overflow`, then the `overflow` exception flag is raised; otherwise it is not raised.

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

Example: Clear the `invalid` exception:

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

Example: Clear all exceptions:

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

Example: Set rounding direction to zero:

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

Example: Set rounding precision to `double`:

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

Turning Off All Warning Messages With `ieee_flags`

Calling `ieee_flags` with an *action* of `clear`, as shown in the following example, resets any uncleared exceptions. Put this call before the program exits to suppress system warning messages about floating-point exceptions at program termination.

Example: Clear all accrued exceptions with `ieee_flags()`:

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

Detecting an Exception With `ieee_flags`

The following example demonstrates how to determine which floating-point exceptions have been raised by earlier computations. Bit masks defined in the system `include` file `f77_floatingpoint.h` are applied to the value returned by `ieee_flags`.

In this example, `DetExcFlg.F`, the `include` file is introduced using the `#include` preprocessor directive, which requires us to name the source file with a `.F` suffix. Underflow is caused by dividing the smallest double-precision number by 2.

Example: Detect an exception using `ieee_flags` and decode it:

```
#include "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
```

Example: Compile and run the preceding example (`DetExcFlg.F`):

```
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%
```

IEEE Extreme Value Functions

The compilers provide a set of functions that can be called to return a special IEEE extreme value. These values, such as *infinity* or *minimum normal*, can be used directly in an application program.

Example: A convergence test based on the smallest number supported by the hardware would look like:

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

The values available are listed in Table 6-2.

Table 6-2 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()

The two NaN values ("quiet" and "signaling") are "unordered" and should not be used in comparisons such as IF(X.ne.r_quiet_nan())THEN... To determine whether some value is a NaN, use the function ir_isnan(r) or id_isnan(d).

The Fortran names for these functions are listed in these man pages:

- libm_double(3f)
- libm_single(3f)
- ieee_functions(3m)

Also see:

- ieee_values(3m)
- The f77_floatingpoint.h header file

Exception Handlers and ieee_handler()

Typical concerns about IEEE exceptions are:

- What happens when an exception occurs?
- How do I use ieee_handler() to establish a user function as an exception handler?
- How do I write a function that can be used as an exception handler?
- How do I locate the exception—where did it occur?

Exception trapping to a user routine begins with the system generating a signal on a floating-point exception. The standard UNIX name for *signal: floating-point exception* is SIGFPE. The default situation on SPARC, Intel, and PowerPC hardware systems is *not* to generate a SIGFPE when an exception occurs. For the system to generate a SIGFPE, exception trapping must first be enabled, usually by a call to ieee_handler().

Establishing an Exception Handler Function

To establish a function as an exception handler, pass the name of the function to ieee_handler(), together with the name of the exception to watch for and the action to take. Once you establish a handler, a SIGFPE signal is generated whenever the particular floating-point exception occurs, and the specified function is called.

The form for invoking `ieee_handler()` is:

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

The routine that calls `ieee_handler()` should also declare:

```
#include 'f77_floatingpoint.h'
```

The special arguments `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, and `SIGFPE_ABORT` are defined in `f77_floatingpoint.h` and can be used to change the behavior of the program for a specific exception:

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

Writing User Exception Handler Functions

What actions your exception handler takes are up to you. However, the routine must be an integer function with three arguments and data types as follows:

- *handler_name*`( sig, sip, uap )`
  - *handler_name* is the name of the integer function.
  - `sig` is an integer.
  - `sip` is a record that has the structure `siginfo.`
  - `uap` is not used.

Example: An exception handler function:

```
INTEGER FUNCTION hand( sig, sip, uap )
INTEGER sig, location
STRUCTURE /fault/
        INTEGER address
        INTEGER trapno
END STRUCTURE
STRUCTURE /siginfo/
        INTEGER si_signo
        INTEGER si_code
        INTEGER si_errno
        RECORD /fault/ fault
END STRUCTURE
RECORD /siginfo/ sip
location = sip.fault.address
... actions you take ...
END
```

If the handler routine enabled by `ieee_handler()` is in Fortran as shown above, it should not make any reference to its first argument (`sig`). This first argument is passed *by value* to the routine and can only be referenced as `loc(sig)`. The value is the signal number.

Detecting an Exception by Handler

The following examples show how to create handler routines to detect floating-point exceptions.

Example: Detect exception and abort:

```
demo% cat DetExcHan.f
    EXTERNAL myhandler
    REAL r / 14.2 /, s / 0.0 /
    i = ieee_handler ('set', 'division', myhandler )
    t = r/s
    END

    INTEGER FUNCTION myhandler(sig,code,context)
    INTEGER sig, code, context(5)
    CALL abort()
    END
demo% f77 -silent DetExcHan.f
demo%  a.out
abort: called
Abort (core dumped)
demo%
```

`SIGFPE` is generated whenever that floating-point exception occurs. When the `SIGFPE` is detected, control passes to the `myhandler` function, which immediately aborts. Compile with `-g` and use `dbx` to find the location of the exception.

Locating an Exception by Handler

Example: Locate an exception (print address) and abort:

```
demo% cat LocExcHan.F
#include "f77_floatingpoint.h"
    EXTERNAL Exhandler
    INTEGER Exhandler, i, ieee_handler
    REAL r / 14.2 /,  s / 0.0 /, t
C  Detect division by zero
    i = ieee_handler( 'set', 'division', Exhandler )
    t = r/s
    END

    INTEGER FUNCTION Exhandler( sig, sip, uap)
    INTEGER sig
    STRUCTURE /fault/
            INTEGER address
    END STRUCTURE
    STRUCTURE /siginfo/
            INTEGER si_signo
            INTEGER si_code
            INTEGER si_errno
            RECORD /fault/ fault
    END STRUCTURE
    RECORD /siginfo/ sip
    WRITE (*,10)  sip.si_signo, sip.si_code, sip.fault.address
10    FORMAT('Signal ',i4,' code ',i4,' at hex address ', Z8 )
    CALL abort()
    END
demo%  f77 -silent -g LocExcHan.F
demo%  a.out
Signal   8 code    3 at hex address    11230
abort: called
Abort (core dumped)
demo%
```

In most cases, knowing the actual *address* of the exception is of little use, except with `dbx`:

```
demo% dbx a.out
(dbx) stopi at 0x11230    Set breakpoint at address
(2) stopi at &MAIN+0x68
(dbx) run                 Run program
Running: a.out
(process id 18803)
stopped in MAIN at 0x11230
MAIN+0x68:fdivs   %f3, %f2, %f2
(dbx) where               Shows the line number of the exception
=>[1] MAIN(), line 7 in "LocExcHan.F"
(dbx) list 7              Displays the source code line
    7   t = r/s
(dbx) cont                Continue after breakpoint, enter handler routine
Signal    8 code    3 at hex address    11230
abort: called
signal ABRT (Abort) in _kill at 0xef6e18a4
_kill+0x8:bgeu    _kill+0x30
Current function is exhandler
   24   CALL abort()
(dbx) quit
demo%
```

Of course, we don't have to go through nearly all this to determine the source line that caused the error... there are easier ways as we will see in a later section. However, this example does serve to show the basics of exception handling.

Disabling All Signal Handlers

By default, some system signal handlers for trapping interrupts, bus errors, segmentation violations, or illegal instructions are automatically enabled.

Although generally you would not want to turn off this default behavior, you can do so by compiling a C program that sets the global C variable `f77_no_handlers` to 1 and linking into your executable program:

```
demo% cat NoHandlers.c
    int  f77_no_handlers=1 ;
demo% cc -c NoHandlers.c
demo% f77 NoHandlers.o MyProgram.f
```

Otherwise, by default, `f77_no_handlers` is 0. The setting takes effect just before execution is transferred to the user program.

This variable is in the global name space of the program; do not use `f77_no_handlers` as the name of a variable anywhere else in the program.

Retrospective Summary

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 you which exceptions were raised but not cleared. This function is automatically called by Fortran programs at normal program termination (`CALL EXIT`). The message typically looks like this; the format varies with each release:

```
Note: IEEE floating-point exception flags raised:
    Division by Zero;
IEEE floating-point exception traps enabled:
    inexact;  underflow;  overflow;  invalid operation;
See the Numerical Computation Guide, ieee_flags(3M),
    ieee_handler(3M)
```

SPARC: Nonstandard Arithmetic

One aspect of standard IEEE arithmetic, called *gradual underflow*, can be manually disabled. When disabled, the program is considered to be running with nonstandard arithmetic.

The IEEE standard for arithmetic specifies a way of handling underflowed results gradually by dynamically adjusting the radix point of the significand. In IEEE floating-point format, the radix point occurs before the significand, and there is an implicit leading bit of 1. Gradual underflow allows the implicit leading bit to be cleared to 0 and shifts the radix point into the significant when the result of a floating-point computation would otherwise underflow. With a SPARC processor this result is not accomplished in hardware but in software. If your program generates many underflows (perhaps a sign of a problem with your algorithm), and you run on a SPARC processor, you may experience a performance loss.

Gradual underflow can be disabled either by compiling with the `-fns` option or by calling the library routine `nonstandard_arithmetic()` from within the program to turn it off— and then calling `standard_arithmetic()` to turn gradual underflow back on.

Note – To be effective, the application's main program must be compiled with `-fns`. See the *Fortran User's Guide*.

For legacy applications, take note that:

- The `standard_arithmetic()` subroutine replaces an earlier routine named `gradual_underflow()`.
- The `nonstandard_arithmetic()` subroutine replaces an earlier routine named `abrupt_underflow()`.

Note – The `-fns` option and the `nonstandard_arithmetic()` library routine are effective only on some SPARC systems. On Intel and PowerPC processors, gradual underflow is performed by the hardware.

`-ftrap=`*mode* Compiler Options

The `-ftrap=`*mode* option enables trapping for floating-point exceptions. If no signal handler has been established by an `ieee_handler()` call, the exception terminates the program with a memory dump core file. See *Fortran User's Guide* for details on this compiler option. For example, to enable trapping for overflow, division by zero, and invalid operations, compile with `-ftrap=common`.

Note – Compile the application's main program with `-ftrap=` for it to be effective.

Floating-Point Exceptions–`f77` *vs* `f90`

Programs compiled by `f77` automatically display a list of accrued floating-point exceptions on program termination. In general, a message results if any one of the invalid, division-by-zero, or overflow exceptions have occurred. Inexact exceptions do not generate messages because they occur so frequently in real programs.

`f90` programs do not automatically report on exceptions at program termination. An explicit call to `ieee_retrospective`(3M) is required.

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.

Debugging IEEE Exceptions

In most cases, the only indication that any floating-point exceptions (such as overflow, underflow, or invalid operation) have occurred is the retrospective summary message at program termination. Locating *where* the exception occurred requires exception that trapping be enabled. This can be done by either compiling with the `-ftrap=common` option or by establishing an exception handler routine with `ieee_handler()`. With exception trapping enabled, run the program from `dbx` or the WorkShop, using the `dbx catch FPE` command to see where the error occurs.

The advantage of recompiling with `-ftrap=common` is that the source code need not be modified to trap the exceptions. However, by calling `ieee_handler()` you can be more selective as to which exceptions to look at.

Example: Recompiling with `-ftrap=common` and using `dbx`:

```
demo% f77 -g -ftrap=common -silent myprogram.f
demo% dbx a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for libF77.so.3
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19739)
signal FPE (floating point divide by zero) in MAIN at line 212 in
file "myprogram.f"
  212   Z = X/Y
(dbx)  print Y
y = 0.0
(dbx)
```

If you find that the program terminates with overflow and other exceptions, you can locate the first overflow specifically by calling `ieee_handler()` to trap just overflows. This requires modifying the source code of at least the main program, as shown in the following example.

Example: Locate an overflow when other exceptions occur:

```
demo% cat myprog.F
#include “f77_floatingpoint.h”
        program myprogram
...
      ier = ieee_handler(‘set’,’overflow’,SIGFPE_ABORT)
...
demo% f77 -g -silent myprog.F
demo% dbx a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for libF77.so.3
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19793)
signal FPE (floating point overflow) in MAIN at line 55 in file
"myprog.F"
   55   w = rmax * 200.                    ! Cause of the overflow
(dbx) cont                                ! Continue execution to completion
 Note: IEEE floating-point exception flags raised:
    Inexact;  Division by Zero;  Underflow;  ! There were other exceptions
 IEEE floating-point exception traps enabled:
    overflow;
 See the Numerical Computation Guide...
execution completed, exit code is 0
(dbx)
```

To be selective, the example introduces the `#include`, which required renaming the source file with a `.F` suffix and calling `ieee_handler()`. You could go further and create your own handler function to be invoked on the overflow exception to do some application-specific analysis, and print intermediary or debug results before aborting.

Further Numerical Adventures

This section addresses some real world problems that involve arithmetic operations that may unwittingly generate invalid, division by zero, overflow, underflow, or inexact exceptions.

For instance, prior to the IEEE standard, if you multiplied two very small numbers on a computer, you could get zero. Most mainframes and minicomputers behaved that way. With IEEE arithmetic, *gradual underflow* expands the dynamic range of computations.

For example, consider a machine with `1.0E-38` as the machine's *epsilon*, the smallest representable value on the machine. Multiply two small numbers:

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

In older arithmetic, you would get `0.0`, but with IEEE arithmetic and the same word length, you get `1.40130E-45`. Underflow tells you that you have an answer smaller than the machine naturally represents. This result is accomplished by "stealing" some bits from the mantissa and shifting them over to the exponent. The result, a *denormalized number*, is less precise in some sense, but more precise in another. The deep implications are beyond this discussion. If you are interested, consult *Computer*, January 1980, Volume 13, Number 1, particularly J. Coonen's article, "Underflow and the Denormalized Numbers."

Most scientific programs have sections of code that are sensitive to roundoff, often in an equation solution or matrix factorization. Without gradual underflow, programmers are left to implement their own methods of detecting the approach of an inaccuracy threshold or else they must abandon the quest for a robust, stable implementation of their algorithm.

For more details on these topics, see the Sun *Numerical Computation Guide*.

Simple Underflow

Some applications actually do a lot of computation very near zero. This is common in algorithms computing residuals or differential corrections. For maximum numerically safe performance, perform the key computations in extended precision arithmetic. If the application is a single-precision application, you can perform key computations in double precision.

Example: A simple dot product computation in single precision:

```
sum = 0
DO i = 1, n
       sum = sum + a(i) * b(i)
END DO
```

If `a(i)` and `b(i)` are very small, many underflows occur. By forcing the computation to double precision, you compute the dot product with greater accuracy and do not suffer underflows:

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

For SPARC systems – You can force a SPARC processor to behave like an older system with respect to underflow (Store Zero) by adding a call to the library routine `nonstandard_arithmetic()` or by compiling the application's main program with the `-fns` option.

Continuing With the Wrong Answer

You might wonder why continue a computation if the answer is clearly wrong. IEEE arithmetic allows you to make distinctions about what kind of wrong answers can be ignored, such as `NaN` or `Inf`. Then decisions can be made based on such distinctions.

For an example, consider a circuit simulation. The only variable of interest (for the sake of argument) from a particular 50-line computation is the voltage. Further, assume that the only values that are possible are +5v, 0, -5v.

It is possible to carefully arrange each part of the calculation to coerce each 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. 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 (SPARC Only)

If two very small numbers are multiplied, the result underflows.

If you know in advance that the operands in a multiplication (or subtraction) may be small and underflow is likely, run the calculation in double precision and convert the result to single precision later.

For example, a dot product loop:

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

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

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

Doing so may also improve performance due to the software resolution of excessive underflows caused by the original loop. However, there is no hard and fast rule here; experiment with your intensely computational code to determine the most profitable solutions.

Porting from Scientific Mainframes

If the application code was originally developed for 64-bit (or 60-bit) mainframes such as CRAY or CDC, you may want to compile these codes with the -dbl option to preserve the expected precision of the original. This option automatically promotes all default REAL variables to DOUBLE PRECISION. This option promotes only undeclared variables or declared as simply REAL; variables declared explicitly REAL*4 will not be promoted. On SPARC, -dbl also promotes variables simply declared DOUBLE to REAL*16 and promotes COMPLEX to COMPLEX DOUBLE. See the *Fortran User's Guide* for details.

To further recreate the original mainframe environment, it is probably preferable to stop on overflows, division by zero, and invalid operations. Compile the main program with -ftrap=common to ensure this.

Porting 7

This chapter discusses the porting of programs from other dialects of Fortran to Sun compilers. VAX VMS Fortran programs compile almost exactly as is with Sun `f77`; this is discussed further in the chapter on VMS extensions in the *Fortran 77 Language Reference*.

Note – Porting issues bear mostly upon Fortran 77 programs. The Sun Fortran 90 compiler, `f90`, incorporates few nonstandard extensions, and these are described in the *Fortran User's Guide*.

Time Functions

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

The following time functions are not supported directly in the Sun Fortran libraries, 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).

The time functions supported in the Sun Fortran library are listed in Table 7-1:

Table 7-1 Sun Fortran Time Functions

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)

For details, see *Fortran Library Reference* or the individual `man` pages for these functions.

The routines listed in Table 7-2 provide compatibility with VMS Fortran system routines `idate` and `time`. To use these routines, you must include the `-lV77` option on the `f77/f90` command line, in which case you also get these VMS versions instead of the standard `f77` versions.

Table 7-2 Summary: VMS Fortran System Routines

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

The error condition subroutine `errsns` is *not* provided, because it is totally specific to the VMS operating system.

Here is a simple example of the use of these time functions (`TestTim.f`):

```
    subroutine startclock
    common / myclock / mytime
    integer mytime, time
    mytime = time()
    return
    end
    function wallclock
    integer wallclock
    common / myclock / mytime
    integer mytime, time, newtime
    newtime = time()
    wallclock = newtime – mytime
    mytime = newtime
    return
    end
    integer wallclock, elapsed
    character*24 greeting
    real dtime, timediff, timearray(2)
c   print a heading
    call fdate( greeting )
    print*,  "Hello, Time Now Is: ",  greeting
    print*,"See how long 'sleep 4' takes, in seconds"
    call startclock
    call system( 'sleep 4' )
    elapsed = wallclock()
    print*, "Elapsed time for sleep 4 was: ", elapsed," seconds"
c   now test the cpu time for some trivial computing
    timediff = dtime( timearray )
    q = 0.01
    do 30 i = 1, 1000
        q = atan( q )
30  continue
    timediff = dtime( timearray )
    print*, "atan(q) 1000 times took: ", timediff ," seconds"
    end
```

Running this program produces the following results:

```
demo% TimeTest
    Hello, Time Now Is: Mon Feb 12 11:53:54 1996
 See how long 'sleep 4' takes, in seconds
 Elapsed time for sleep 4 was:   5 seconds
 atan(q) 1000 times took:     2.26550E-03 seconds
demo%
```

Formats

Some f77 format edit descriptors may behave differently on other systems. Here are some format specifiers that f77 treats differently than some other implementations:

- A—Used with character type data elements. In Fortran, this specifier worked with any variable type. f77 supports the older usage, up to four characters to a word.
- $—Suppresses newline character output.
- R—Sets an arbitrary radix for the I formats that follow in the descriptor.
- SU—Selects 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 )
```

Carriage-Control

Fortran carriage-control grew out of the capabilities of the equipment used when Fortran was originally developed. For similar historical reasons, an operating system derived from the UNIX operating system, does not have Fortran carriage-control, but you can simulate it in two ways.

- *(f77 only)* For simple jobs, use `OPEN(N, FORM='PRINT')` to enable 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 carriage-control conventions into the UNIX carriage-control format (see the `asa` (1) man page) before printing files with the `lpr` command.

Working With Files

Early Fortran systems did not use named files, but did provide a command line mechanism to equate actual file names with internal unit numbers. This facility can be emulated in a number of ways, including standard UNIX redirection:

Example: Redirecting `stdin` to `redir.data` (using `csh`(1)):

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

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

demo% f77 -silent -o redir redir.f      ← The compilation step
demo% redir < redir.data      ← Run with redirection reads data file
  9 9.90000
demo%
```

See Chapter 2, *Fortran Input/Output*, for more on redirection and working with files.

7

Data Representation

The *Fortran 77 Language Reference* and the *Sun Numerical Computation Guide* discuss in detail the hardware representation of data objects in Fortran. Differences between data representations across systems and hardware platforms usually generate the most significant portability problems.

The following issues should be noted:

- Sun adheres to the IEEE Standard 754 for floating-point arithmetic. Therefore, the first four bytes in a `REAL*8` are not the same as in a `REAL*4`.
- The default sizes for reals, integers, and logicals are described in the Fortran standard, except:
  - when the `-i2` flag is used, which shrinks integers and logicals to two bytes, but leaves reals as four bytes
  - when the `–dbl` or `–r8` flags are used to promote integers, logicals and reals to eight bytes
- Character variables can be freely mixed and equivalenced to variables of other types, but be careful of potential alignment problems.
- `f77` IEEE floating-point arithmetic does raise exceptions on overflow or divide by zero but does not signal `SIGFPE` or trap by default. It does deliver IEEE indeterminate forms in cases where exceptions would otherwise be signaled. This is explained in the *Floating Point Arithmetic* chapter of this Guide.
- 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.

Hollerith Data

Many “dusty-deck” Fortran applications store Hollerith ASCII data into numerical data objects. With the 1977 Fortran standard, the `CHARACTER` data type was provided for this purpose and its use is recommended. You can still initialize variables with the older Fortran Hollerith (*n*`H`) feature, but this is not

standard practice. Table 7-3 indicates the maximum number of characters that will fit into certain data types. (In this table, boldfaced data types indicate default types subject to promotion by `-dbl`, `-r8`, or `-xtypemap=` .)

Table 7-3 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
`INTEGER*8`	8	8	8	8	8
`LOGICAL`	4	2	4	4	8
`LOGICAL*1`	1	1	1	1	1
`LOGICAL*8`	8	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

When 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.

The storage is allocated with both options, but it is unavailable in normal Fortran with `-r8`.

Example: Initialize variables with Hollerith:

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

demo% f77 -silent -o FourA8 FourA8.f
demo% FourA8
abcdefghijklmnopqrstuvwxyz012345!
demo%
```

If you pass Hollerith constants as arguments, or if you use them in expressions or comparisons, they are interpreted as character-type expressions.

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

Example:

```
        program respond
        integer yes, no
        integer ask
        data yes, no / 3hyes, 2hno /

        if ( ask() .eq. yes ) then
            print *, 'You may proceed!'
        else
            print *, 'Request Rejected!'
        endif
        end

        integer function ask()
        double precision solaris, response
        integer yes, no
        data yes, no / 3hyes, 2hno /
        data solaris/ 7hSOLARIS/
10      format( "What system? ", $ )
20      format( a8 )

        write( 6, 10 )
        read ( 5, 20 ) response
        ask = no
        if ( response .eq. solaris ) ask = yes
        return
        end
```

Nonstandard Coding Practices

As a general rule, porting an application program from one system and compiler to another can be made easier by eliminating any nonstandard coding. Optimizations or work-arounds that were successful on one system may only serve to obscure and confuse compilers on other systems. In particular, optimized hand-tuning for one particular architecture may turn out to cause degradations in performance elsewhere. This is discussed later in the chapters on performance and tuning. However, the following issues are worth considering with regards to porting in general.

Uninitialized Variables

Some systems automatically initialize local and COMMON variables to zero or some not-a-number value. However, there is no standard practice, and programs should not make assumptions regarding the initial value of any variable. To assure maximum portability, a program should initialize all variables.

Aliasing Across Calls

Aliasing occurs when the same storage address is referenced by more than one name. This happens when actual arguments to a subprogram overlap between themselves or between COMMON variables within the subprogram. For example, arguments X and Z refer to the same storage locations, as do B and H:

```
COMMON /INS/B(100)
REAL S(100), T(100)
...
CALL SUB(S,T,S,B,100)
...
SUBROUTINE SUB(X,Y,Z,H,N)
REAL X(N),Y(N),Z(N),H(N)
COMMON /INS/B(100)
...
```

Aliasing in this manner should be avoided in all portable code. The results on some systems and with higher optimization levels could be unpredictable.

Obscure Optimizations

Legacy codes may contain source-code restructurings of ordinary computational DO loops intended to cause older vectorizing compilers to generate optimal code for a particular architecture. In most cases, these restructurings are no longer needed and may degrade the portability of a program. Two common restructurings are strip-mining and loop unrolling.

Strip-Mining

Fixed-length vector registers on some architectures led programmers to manually “strip-mine” the array computations in a loop into segments:

```
REAL TX(0:63)
...
DO IOUTER = 1,NX,64
   DO IINNER = 0,63
      TX(IINNER) = AX(IOUTER+IINNER) * BX(IOUTER+IINNER)/2.
      QX(IOUTER+IINNER) = TX(IINNER)**2
   END DO
END DO
```

Strip-mining is no longer appropriate with modern compilers; the loop can be written much less obscurely as:

```
DO IX = 1,N
  TX = AX(I)*BX(I)/2.
  QX(I) = TX**2
END DO
```

Loop Unrolling

Unrolling loops by hand was a typical source-code optimization technique before compilers were available that could perform this restructuring automatically. A loop written as:

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

should be rewritten the way it was originally intended:

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

Troubleshooting

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

Results Are Close, but Not Close Enough

Try the following:

- Pay attention to the size and the engineering units. Numbers very close to zero can appear to be different, but the difference is not significant. 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 the mathematics involves many trigonometric functions. These functions are much more complicated than one might think, and the standard defines only the basic arithmetic functions. There can be subtle differences, even between IEEE machines. Review the chapter *Floating-Point Arithmetic*, in this Guide.

- Try running with a `call nonstandard_arithmetic()`. Doing so can also improve performance considerably, and make your Sun workstation behave more like a VAX. If you have access to a VAX or some other system, run it there, also. It is quite common for many numerical applications to produce slightly different results on each floating-point implementation.

- Check for `NaN`, `+Inf`, and other signs of probable errors. See the chapter *Floating-Point Arithmetic* in this Guide, or the `man` page `ieee_handler`(3m) for instructions on how to trap the various exceptions. On most machines, these exceptions simply abort the run.

- Two numbers can differ by 6 x 10^{29} and still have the same floating-point form. Here is an example of different numbers, with the same representation:

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

The output is:

```
99,999,990 x 10^29 = 0.99999993E+37 = 7cf0bdc1
99,999,996 x 10^29 = 0.99999993E+37 = 7cf0bdc1
```

In this example, the difference is 6 x 10^{29}. The reason for this indistinguishable, wide gap is that in IEEE single-precision arithmetic, you are guaranteed only six decimal digits for any one decimal-to-binary conversion. You may be able to convert seven or eight digits correctly, but it depends on the number.

Program Fails without Warning

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

- Compile with minimal optimization (`-O1`). If the program then works, compile only selective routines with higher optimization levels.
- Understand that optimizers must make assumptions about the program. Nonstandard coding or constructs can cause problems. Almost no optimizer handles all programs at all levels of optimization.

Performance Profiling 8

This chapter describes how to measure and display program performance. Knowing where a program is spending most of its compute cycles and how efficiently it uses system resources is a prerequisite for performance tuning.

The `time` Command

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

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

```
demo% time myprog
   The Answer is: 543.01
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
demo%
```

The interpretation is:

user system wallclock resources memory I/O paging

- *user* – 6.5 seconds in user code, approximately
- *system* – 17.1 seconds in system code for this task, approximately
- *wallclock* – 1 minute 16 seconds to complete
- *resources* – 31% of system resources dedicated to this program

- *memory* – 11 kilobytes of shared program memory, 21 kilobytes of private data memory
- *I/O* – 354 reads, 210 writes
- *paging* – 135 page faults, 0 swapouts

Multiprocessor Interpretation of `time` *Output*

Timing results are interpreted in a different way when the program is run in parallel in a multiprocessor environment. Since `/bin/time` accumulates the user time on different threads, only wall clock time is used.

Since the user time displayed includes the time spent on all the processors, it can be quite large and is not a good measure of performance. A better measure is the real time, which is the wall clock time. This also means that to get an accurate timing of a parallelized program it must be run on a quiet system dedicated to just your program.

The `gprof` *Profiling Command*

The `gprof` (1) command provides a detailed postmortem analysis of program timing at the subprogram level, including how many times a subprogram was called, who called it and whom it called, and how much time was spent in the routine and by the routines it called.

To enable `gprof` profiling, compile and link the program with the `-pg` option:

```
demo% f77 -o Myprog -fast -pg Myprog.f ...etc
...
demo% gprof Myprog
```

The program must complete normally for `gprof` to obtain meaningful timing information.

At program termination, the file `gmon.out` is automatically written in the working directory. This file contains the profiling data that will be interpreted by `gprof`.

Invoking `gprof` produces a report on standard output. An example is shown on the next pages. Not only the routines in your program are listed but also the library procedures and the routines they call.

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. (The `gprof -b` option eliminates the explanatory text; see the `gprof`(1) man page for other options that can be used to limit the amount of output generated.)

In the graph profile, each procedure (subprogram, procedure) is presented in a call-tree representation. The line representing a procedure in its call-tree is called the *function line*, and is identified by an index number in the leftmost column, within square brackets; the lines above it are the *parent lines*; the lines below it, the *descendant* lines.

```
-----------------------------------------------------------------
                parent line                     caller 1
                parent line                     caller 2
                    ....
[n]  time       function line                   function name
                descendant line                 called 1
                descendant line                 called 2
                    ....
-----------------------------------------------------------------
```

The call graph profile is followed by a flat profile that provides a routine-by-routine overview. An (edited) example of `gprof` output follows.

Note – User-defined functions appear with their Fortran names followed by an underscore. Library routines appear with leading underscores.

The call graph profile:

```
granularity: each sample hit covers 2 byte(s) for 0.08% of 12.78 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name    index
                                  called/total       children

                0.00       12.66       1/1           main [1]
[3]     99.1    0.00       12.66       1         MAIN_ [3]
                0.92       10.99    1000/1000        differ_ [4]
                0.62        0.00    2000/2001        code_ [9]
                0.11        0.00    1000/1000        shock_ [11]
                0.02        0.00    1000/1000        bndry_ [14]
                0.00        0.00       1/1           init_ [24]
                0.00        0.00       2/2           output_ [40]
                0.00        0.00       1/1           input_ [47]

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

                0.92       10.99    1000/1000        MAIN_ [3]
[4]     93.2    0.92       10.99    1000         differ_ [4]
                1.11        4.52    3000/3000        deriv_ [7]
                1.29        2.91    3000/6000        cheb1_ [5]
                1.17        0.00    3000/3000        dissip_ [8]

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

                1.29        2.91    3000/6000        deriv_ [7]
                1.29        2.91    3000/6000        differ_ [4]
[5]     65.7    2.58        5.81    6000         cheb1_ [5]
                5.81        0.00    6000/6000        fftb_ [6]
                0.00        0.00     128/321         cos [21]
                0.00        0.00     128/192         __sin [279]

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

                5.81        0.00    6000/6000        cheb1_ [5]
[6]     45.5    5.81        0.00    6000         fftb_ [6]
                0.00        0.00      64/321         cos [21]
                0.00        0.00      64/192         __sin [279]

-----------------------------------------------
...
```

The flat profile overview:

```
granularity: each sample hit covers 2 byte(s) for 0.08% of 12.84
seconds

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 45.2       5.81     5.81     6000     0.97     0.97  fftb_ [6]
 20.1       8.39     2.58     6000     0.43     1.40  cheb1_ [5]
  9.1       9.56     1.17     3000     0.39     0.39  dissip_ [8]
  8.6      10.67     1.11     3000     0.37     1.88  deriv_ [7]
  7.1      11.58     0.92     1000     0.92    11.91  diffr_ [4]
  4.8      12.20     0.62     2001     0.31     0.31  code_ [9]
  2.5      12.53     0.33    69000     0.00     0.00  __exp [10]
  0.9      12.64     0.11     1000     0.11     0.11  shock_ [11]
...
```

- *Function Line*

The function line [5] in the preceding example reveals that:

- `cheb1` was called 6000 times— 3000 from `deriv`, 3000 from `diffr`.
- 2.58 seconds were spent in `cheb1` itself.
- 5.81 seconds were spent in routines called by `cheb1`.
- 65.7% of the execution time of the program was within `cheb1`.

- *Parent Lines*

  The parent lines above [5] indicate that `cheb1` was called from two routines, `deriv` and `diffr`. The timings on these lines show how much time was spent in `cheb1` when it was called from each of these routines.

- *Descendant Lines*

  The lines below the function line indicate the routines called from `cheb1`, `fftb`,`sin`, and `cos`. The library sine function is called indirectly.

- *Flat Profile*

  Function names appear on the right. The profile is sorted by percentage of total execution time.

Overhead Considerations

Profiling (compiling with the `-pg` option) may greatly increase the running time of a program. This is due to the extra overhead required to clock program performance and subprogram calls. Profiling tools like `gprof` attempt to subtract an approximate overhead factor when computing relative runtime percentages. All other timings shown may not be accurate due to UNIX and hardware timekeeping inaccuracies.

Programs with short execution times are the most difficult to profile because the overhead may be a significant fraction of the total execution time. The best practice is to choose input data for the profiling run that will result in a realistic test of the program's performance. If this is not possible, consider enclosing the main computational part of the program within a loop that effectively runs the program *N* times. Estimate actual performance by dividing the profile results by *N*.

The Fortran library includes two routines that return the total time used by the calling process. See `dtime`(3F) and `etime`(3F).

Missing Profile Libraries

If the profiling libraries are not installed when 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.

The `tcov` *Profiling Command*

The `tcov` (1) command, when used with programs compiled with the `-a`, `-xa`, or `-xprofile=tcov` options, produces a statement-by-statement profile of the source code showing which statements executed and how often. It also gives a summary of information about the basic block structure of the program.

There are two implementations of `tcov` coverage analysis. The original `tcov` is invoked by the `-a` or `-xa` compiler options. Enhanced statement level coverage is invoked by the `-xprofile=tcov` compiler option and the `-x tcov` option. In either case, the output is a copy of the source files annotated with statement execution counts in the margin. Although these two versions of `tcov` are essentially the same as far as the Fortran user is concerned (most of the enhancements apply to C++ programs), there will be some performance improvement with the newer style.

"Old Style" `tcov` *Coverage Analysis*

Compile the program with the `-a` (or `-xa`) option. This produces the file `$TCOVDIR/`*file*`.d` for each source `.f` file in the compilation. (If environment variable `$TCOVDIR` is not set at compile time, the `.d` files are stored in the current directory.)

Run the program (execution must complete normally). This produces updated information in the `.d` files. To view the coverage analysis merged with the individual source files, run `tcov` on the source files. The annotated source files are named `$TCOVDIR/`*file*`.tcov` for each source file.

The output produced by `tcov` shows the number of times each statement was actually executed. Statements that were not executed are marked with `####->` to the left of the statement.

Here is a simple example:

```
demo% f77 -a -o onetwo -silent one.f two.f
demo% onetwo
     ... output from program
demo% tcov one.f two.f
demo% cat one.tcov two.tcov
           program one
      1 -> do i=1,10
     10 -> call two(i)
           end do
      1 -> end

         Top 10 Blocks
         Line   Count
            3      10
            2       1
            5       1

        3 Basic blocks in this file
        3 Basic blocks executed
    100.00 Percent of the file executed
           12 Total basic block executions
         4.00 Average executions per basic block

           subroutine two(i)
     10 -> print*, "two called", i
           return
           end

         Top 10 Blocks
         Line   Count
            2      10

        1 Basic blocks in this file
        1 Basic blocks executed
    100.00 Percent of the file executed
           10 Total basic block executions
        10.00 Average executions per basic block
demo%
```

"New Style" Enhanced `tcov` *Analysis*

To use new style `tcov`, compile with `-xprofile=tcov`. When the program is run, coverage data is stored in *program*`.profile/tcovd`, where *program* is the name of the executable file. (If the executable were `a.out`, `a.out.profile/tcovd` would be created.)

Run `tcov -x` *dirname source_files* to create on *file*`.tcov` in the current directory the coverage analysis merged with each source file.

Running a simple example:

```
demo% f77 -o onetwo -silent -xprofile=tcov one.f two.f
demo% onetwo
     ... output from program
demo% tcov -x onetwo.profile one.f two.f
demo% cat one.f.tcov two.f.tcov
           program one
      1 -> do i=1,10
     10 -> call two(i)
           end do
      1 -> end
       .....etc
demo%
```

Environment variables `$SUN_PROFDATA` and `$SUN_PROFDATA_DIR` can be used to specify where the intermediary data collection files are kept. These are the `*.d` and `tcovd` files created by old and new style `tcov`, respectively.

Each subsequent run accumulates more coverage data into the *myprog*`.tcovd` file. Data for each object file is zeroed out the first time the program is executed after the corresponding source file has been recompiled. Data for the entire program is zeroed out by removing the *myprog*`.tcovd` file.

These environment variables can be used to separate the collected data from different runs. With these variables set, the running program writes execution data to the files in `$SUN_PROFDATA_DIR/$SUN_PROFDATA/`. Similarly, the directory that `tcov` reads is specified by `tcov -x $SUN_PROFDATA`. If `$SUN_PROFDATA_DIR` is set, `tcov` will prepend it, looking for files in `$SUN_PROFDATA_DIR/$SUN_PROFDATA/`, and not the working directory. For the details, see the `tcov`(1) `man` page.

8

I/O Profiling

You can obtain a report about how much data was transferred by your program. For each Fortran 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 an I/O profiling report, insert calls to the library routines `start_iostats` and `end_iostats` around the parts of the program you wish to measure. (A call to `end_iostats` is required if the program terminates with an `END` or `STOP` statement rather than a `CALL EXIT`.)

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 explicitly reopen these units after the call to `start_iostats` without first closing them for monitoring.

Example: Profile `stdin`, `stdout`, and `stderr`:

```
EXTERNAL start_iostats
…
CALL start_iostats
OPEN(5)
OPEN(6)
OPEN(0)
```

If you want to measure only part of the program, call `end_iostats` to stop the process. A call to `end_iostats` may also be required if your program terminates with an `END` or `STOP` statement rather than `CALL EXIT`.

The program must be compiled with the `-pg` option. When the program terminates, the I/O profile report is produced on the file *name*`.io_stats` . (*name* is the name of the executable file).

Here is an example:

```
demo% f77 -o myprog -pg -silent myprog.f
demo% myprog
    ... output from program
demo% cat myprog.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    No
                             0         0         0         0
      5           stdin      2         8         4         0    No
                             1         8         8         0
      6          stdout      0         0         0         0    No
                             0         0         0         0
     19         fort.19      8        48         6     4.276    No
                             4        48        12         0
     20         fort.20      8        48         6     4.276    No
                             4        48        12         0
     21         fort.21      8        48         6     4.276    No
                             4        48        12         0
     22         fort.22      8        48         6     4.276    No
                             4        48        12         0

             OUTPUT REPORT
1. unit              5. output data            6. blk size  7. fmt  8. direct
            cnt     total       avg   std dev                          (rec len)
-------------------------------------------------------------------------------
      0      4        40        10         0           -1     Yes     seq
             1        40        40         0
      5      0         0         0         0           -1     Yes     seq
             0         0         0         0
      6     26       248     9.538      1.63           -1     Yes     seq
             6       248     41.33     3.266
     19      8        48         6     4.276       500548     Yes     seq
             4        48        12         0
     20      8        48         6     4.276       503116      No     seq
             4        48        12         0
     21      8        48         6     4.276       503116     Yes     dir
             4        48        12         0                         (   12)
     22      8        48         6     4.276       503116      No     dir
             4        48        12         0                         (   12)
     …
```

Each pair of lines in the report displays information about an I/O unit. There is one section showing input operations and another for output. The first line of a pair displays statistics on the number of data elements transferred before the unit was closed. The second row of statistics is based on the number of I/O statements processed.

In the example, there were 6 calls to write a total of 26 data elements to standard output. A total of 248 bytes were transferred. The display also shows the average and standard deviation in bytes transferred per I/O statement (9.538 and 1.63, respectively), and the average and standard deviation per I/O statement call (42.33 and 3.266, respectively).

The input report also contains a column to indicate whether a unit was memory mapped or not. If mapped, the number of `mmap()` calls is recorded in parentheses in the second row of the pair.

The output report indicates block sizes, formatting, and access type. A file opened for direct access shows its defined record length in parentheses in the second row of the pair.

Note – Compiling with environment variable `LD_LIBRARY_PATH` set may disable I/O profiling, which relies on its profiling I/O library being in a standard location.

Performance and Optimization 9

This chapter considers some optimization techniques that may improve the performance of numerically intense Fortran programs. Proper use of algorithms, compiler options, library routines, and coding practices can bring significant performance gains. This discussion does not discuss cache, I/O, or system environment tuning. Parallelization issues are treated in the next chapter.

Some of the issues considered here are:

- Compiler options that may improve performance
- Compiling with feedback from runtime performance profiles
- Use of optimized library routines for common procedures
- Coding strategies to improve performance of key loops

The subject of optimization and performance tuning is much too complex to be treated exhaustively here. However, this discussion should provide the reader with a useful introduction to these issues. A list of books that cover the subject much more deeply appears at the end of the chapter.

Optimization and performance tuning is an art that depends heavily on being able to determine *what* to optimize or tune.

9

Choice of Compiler Options

Choice of the proper compiler options is the first step in improving performance. Sun compilers offer a wide range of options that affect the object code. In the default case, where no options are explicitly stated on the compile command line, most options are *off*. To improve performance, these options must be explicitly selected.

Performance options are normally off by default because most optimizations force the compiler to make assumptions about a user's source code. Programs that conform to standard coding practices and do not introduce hidden side effects should optimize correctly. However, programs that take liberties with standard practices may run afoul of some of the compiler's assumptions. The resulting code may run faster, but the computational results may not be correct.

Recommended practice is to first compile with all options off, verify that the computational results are correct and accurate, and use these results and performance profile as a baseline. Then, proceed in steps— recompiling with additional options and comparing execution results and performance against the baseline. If numerical results change, the program may have questionable code, which needs careful analysis to locate and reprogram.

If performance does not improve significantly, or degrades, as a result of adding optimization options, the coding may not provide the compiler with opportunities for further performance improvements. The next step would then be to analyze and restructure the program at the source code level to achieve better performance.

Performance Option Reference

The compiler options listed in Table 9-1 provide the user with a repertoire of strategies to improve the performance of a program over default compilation. Only some of the compilers' more potent performance options appear below. A more complete list can be found in the *Fortran User's Guide*.

Table 9-1 Some Effective Performance Options

Action	**Option**
Use various optimization options together	`-fast`
Set compiler optimization level to *n*	`-O`*n* (`-O = -O3`)
Specify target hardware	`-xtarget=`*sys*
Optimize using performance profile data (with `-O5`)	`-xprofile=use`
Unroll loops by *n*	`-unroll=`*n*
Permit simplifications and optimization of floating-point	`-fsimple=1|2`
Perform dependency analysis to optimize loops	`-depend`

Some of these options will increase compilation time because they invoke a deeper analysis of the program. Some options work best when routines are collected into files along with the routines that call them (rather than splitting each routine into its own file); this allows the analysis to be global.

`-fast`

This single option selects a number of performance options that, working together, produce object code optimized for execution speed without an excessive increase in compilation time.

The options selected by `-fast` are subject to change from one release to another, and not all are available on each platform:

- `-xtarget=native` – generates code optimized for the host architecture
- `-O4` – sets optimization level
- `-libmil` – inlines calls to some simple library functions
- `-fsimple=1` – simplifies floating-point code (*SPARC only*)
- `-dalign` – uses faster, double word loads and stores (*SPARC only*)
- `-xlibmopt` – use optimized `libm` math library (*SPARC, PowerPC only*)
- `-fns -ftrap=%none` – turns off all trapping

- `-depend` – analyze loops for data dependencies (*SPARC only*)
- `–nofstore` – disables forcing precision on expressions (*Intel only*)

`-fast` provides a quick way to engage much of the optimizing power of the compilers. Each of the composite options may be specified individually, and each may have side effects to be aware of (discussed in the *Fortran User's Guide)*. Following `-fast` with additional options adds further optimizations. For example:

```
f77 -fast -O5 ...
```

sets the optimization to level 5 instead of 4.

Note – `-fast` includes `-dalign` and `-native`. These options may have unexpected side-effects for some programs.

`-O`*n*

No compiler optimizations are performed by the compilers unless a `-O` option is specified explicitly (or implicitly with macro options like `-fast`). In nearly all cases, specifying an optimization level for compilation improves program execution performance. On the other hand, higher levels of optimization increase compilation time and may significantly increase code size.

For most cases, level `-O3` is a good balance between performance gain, code size, and compilation time. Level `-O4` adds automatic inlining of calls to routines contained in the same source file as the caller routine, among other things. Level `-O5` adds more aggressive optimization techniques that would not be applied at lower leves. In general, levels above `-O3` should be specified only to those routines that make up the most compute-intensive parts of the program and thereby have a high certainty of improving performance. (There is no problem linking together parts of a program compiled with different optimization levels.)

SPARC: Optimization With Runtime Profile Feedback

The compiler applies its optimization strategies at level `O3` and above much more efficiently if combined with `-xprofile=use`. With this option (available only on SPARC processors), the optimizer is directed by a runtime execution profile produced by the program (compiled with `-xprofile=collect`) with

typical input data. The feedback profile indicates to the compiler where optimization will have the greatest effect. This may be particularly important with -O5. Here's a typical example of profile collection with higher optimization levels:

```
demo% f77 -o prg -fast -xprofile=collect prg.f ...
demo% prg
demo% f77 -o prgx -fast -O5 -xprofile=use:prg.profile prg.f ...
demo% prgx
```

The first compilation above generates an executable that produces statement coverage statistics when run. The second compilation uses this performance data to guide the optimization of the program.

(See *Fortran User's Guide* and *Performance Profiling Tools* for details on -xprofile options.)

-dalign

With -dalign the compiler is able to generate double-word load/store instructions whenever possible. Programs that do much data motion may benefit significantly when compiled with this option. (It is one of the options selected by -fast.) The double-word instructions are almost twice as fast as the equivalent single word operations.

However, users should be aware that using -dalign (and therefore -fast) may cause problems with some programs that have been coded expecting a specific alignment of data in COMMON blocks. With -dalign, the compiler may add padding to ensure that all double (and quad) precision data (either REAL or COMPLEX) are aligned on double word boundaries, with the result that:

- COMMON blocks may be larger than expected due to added padding
- All program units sharing COMMON must be compiled with -dalign if any one of them is compiled with -dalign

For example, a program that writes data by aliasing an entire COMMON block of mixed data types as a single array may not work properly with -dalign because the block will be larger (due to padding of double and quad precision variables) than the program expects.

SPARC: `-depend` *(`f77` only)*

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

`-fsimple=2` *(`f77` only)*

Unless directed to, the compiler does not attempt to simplify floating-point computations (this is the default, `-fsimple=0`). With the `-fast` option, `-fsimple=1` is used and some conservative assumptions are made. Adding `-fsimple=2` enables the optimizer to make further simplifications with the understanding that this may cause some programs to produce slightly different results due to rounding effects. If `-fsimple` level 1 or 2 is used, all program units should be similarly compiled to insure consistent numerical accuracy,

`-unroll=`*n*

Unrolling short loops with long iteration counts can be profitable for some routines. However, unrolling can also increase program size and may even degrade performance of other loops. With *n*=1, the default, no loops are unrolled automatically by the optimizer. With *n* greater than 1, the optimizer attempts to unroll loops up to a depth of *n*. If a `DO` loop with a variable loop limit can be unrolled, both an unrolled version and the original loop are compiled. A runtime test on iteration count determines whether or not executing the unrolled loop is inappropriate. Loop unrolling, especially with simple one or two statement loops, increases the amount of computation done per iteration and provides the optimizer better opportunities to schedule registers and simplify operations. The tradeoff between number of iterations, loop complexity, and choice of unrolling depth is not easy to determine, and some experimentation may be needed.

The example that follows shows how a simple loop might be unrolled to a depth of four with `-unroll=4` (the source code is not changed with this option):

Original Loop:

```
    DO I=1,20000
       X(I) = X(I) + Y(I)*A(I)
    END DO
```

Unrolled by 4 *compiles as*:

```
    DO I=1, 19997,4
       TEMP1 = X(I) + Y(I)*A(I)
       TEMP2 = X(I+1) + Y(I+1)*A(I+1)
       TEMP3 = X(I+2) + Y(I+2)*A(I+2)
       X(I+3) = X(I+3) + Y(I+3)*A(I+3)
       X(I) = TEMP1
       X(I+1) = TEMP2
       X(I+2) = TEMP3
    END DO
```

This example shows a simple loop with a fixed loop count. The restructuring is more complex with variable loop counts.

`-xtarget=`*system*

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

The *Fortran User's Guide* lists all the system names recognized by `-xtarget=`. For any given system name (for example, `ss1000`, for SPARC Server 1000), `-xtarget` expands into a specific combination of `-xarch`, `-xcache`, and `-xchip` that properly matches that system. The optimizer uses these specifications to determine strategies to follow and instructions to generate.

The special setting `-xtarget=native` enables the optimizer to compile code targeted at the host system (the system doing the compilation). This is obviously useful when compilation and execution are done on the same system. When the execution system is not known, it is desirable to compile for a *generic* architecture, therefore `-xtarget=generic` is the default, although this may produce suboptimal performance.

Other Performance Strategies

Assuming that you have experimented with using a variety of optimization options, compiling your program and measuring actual runtime performance, the next step might be to look closely at the Fortran source program to see what further tuning can be tried.

Focusing on just those parts of the program that use most of the compute time, you might consider the following strategies:

- Replace handwritten procedures with calls to equivalent optimized libraries
- Remove I/O, calls, and unnecessary conditional operations from key loops
- Eliminate aliasing that might inhibit optimization
- Rationalize tangled, spaghetti-like code to use block `IF`

These are some of the good programming practices that tend to lead to better performance. It is possible to go further, hand-tuning the source code for a specific hardware configuration. However, these attempts may only further obscure the code and make it even more difficult for the compiler's optimizer to achieve significant performance improvements. Excessive hand-tuning of the source code may hide the original intent of the procedure and could have a significantly detrimental effect on performance for different architectures.

• Use Optimized Libraries

In most situations, optimized commercial or shareware libraries perform standard computational procedures far more efficiently than you could by coding them by hand.

For example, the Sun Performance Library™ is a suite of highly optimized mathematical subroutines based on the standard LAPACK, BLAS, FFTPACK, VFFTPACK, and LINPACK libraries. Performance improvement using these routines can be significant when compared with hand coding.

• *Eliminate Performance Inhibitors*

Use the profiling techniques described in the previous chapter to identify the key computational parts of the program. Then, carefully analyze the loop or loop nest to eliminate coding that might either inhibit the optimizer from generating optimal code or otherwise degrade performance. See also the chapter on Porting. Many of the nonstandard coding practices that make portability difficult may also inhibit optimization by the compiler.

Reprogramming techniques that improve performance are dealt with in more detail in some of the reference books listed at the end of the chapter. Three major approaches are worth mentioning here:

• *Remove I/O From Key Loops*

I/O within a loop or loop nest enclosing the significant computational work of a program will seriously degrade performance. The amount of CPU time spent in the I/O library may be a major portion of the time spent in the loop. (I/O also causes process interrupts, thereby degrading program throughput.) By moving I/O out of the computation loop wherever possible, the number of calls to the I/O library can be greatly reduced.

• *Eliminate Subprogram Calls*

Subroutines called deep within a loop nest may get called thousands of times. Even if the time spent in each routine per call is small, the total effect may be substantial. Also, subprogram calls inhibit optimization of the loop that contains them because the compiler cannot make assumptions about the state of registers over the call.

Automatic inlining of subprogram calls (using `-inline=`*x,y,..z*, or `-O4`) is one way to let the compiler replace the actual call with the subprogram itself (*pulling* the subprogram into the loop). The subprogram source code for the routines that are to be inlined be must be found in the same file as the calling routine.

There are other ways to eliminate subprogram calls:

- Use statement functions. If the external function being called is a simple math function, it may be possible to rewrite the function as a statement function or set of statement functions. Statement functions are compiled in-line and can be optimized.
- Push the loop into the subprogram. That is, rewrite the subprogram so that it can be called fewer times (outside the loop) and operate on a vector or array of values per call.

• *Rationalize Tangled Code*

Complicated conditional operations within a computationally intensive loop can dramatically inhibit the compiler's attempt at optimization. In general, a good rule to follow is to eliminate all arithmetic and logical `IF`'s, replacing them with block `IF`'s:

```
Original Code:
    IF(A(I)-DELTA) 10,10,11
10  XA(I) = XB(I)*B(I,I)
    XY(I) = XA(I) - A(I)
    GOTO 13
11  XA(I) = Z(I)
    XY(I) = Z(I)
    IF(QZDATA.LT.0.) GOTO 12
    ICNT = ICNT + 1
    ROX(ICNT) = XA(I)-DELTA/2.
12  SUM = SUM + X(I)
13  SUM = SUM + XA(I)

Untangled Code:
    IF(A(I).LE.DELTA) THEN
      XA(I) = XB(I)*B(I,I)
      XY(I) = XA(I) - A(I)
    ELSE
      XA(I) = Z(I)
      XY(I) = Z(I)
      IF(QZDATA.GE.0.) THEN
        ICNT = ICNT + 1
        ROX(ICNT) = XA(I)-DELTA/2.
      ENDIF
      SUM = SUM + X(I)
    ENDIF
    SUM = SUM + XA(I)
```

Using block `IF` not only improves the opportunities for the compiler to generate optimal code, it also improves readability and assures portability.

Further Reading

The following reference books provide more details:

- *Fortran 77 Language Reference*, Sun Microsystems, Inc.
- *Numerical Computation Guide*, Sun Microsystems, Inc.
- *Performance Profiling Tools*, Sun Microsystems, 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*, John Levesque, Joel Williamson, Academic Press, 1989
- *High Performance Computing*, Kevin Dowd, O'Reilly & Associates, 1993

9

Parallelization 10

This chapter presents an overview of multiprocessor parallelization and describes the capabilities of Sun's Fortran compilers. Implementation differences between `f77` and `f90` are noted.

Note – Parallelization features are only available on SPARC platforms running Solaris 2.x, and require a Sun WorkShop license.

Introduction

Parallelizing (or *multithreading*) an application recasts the compiled program to run on a multiprocessor system. Parallelization enables single tasks, such as a `DO` loop, to run over multiple processors with a potentially significant execution speedup.

Before an application program can be run efficiently on a multiprocessor system like the SPARCstation 10 or SPARCcenter 2000, it needs to be multithreaded. That is, tasks that can be performed in parallel need to be identified and reprogrammed to distribute their computations.

Multithreading an application can be done manually by making appropriate calls to the `libthread` primitives. However, a significant amount of analysis and reprogramming may be required. (See the Solaris *Multithreaded Programming Guide* for more information.)

Sun compilers can automatically generate multithreaded object code that to run on multiprocessor systems. The Fortran compilers focus on `DO` loops as the primary language element supporting parallelism. Parallelization distributes the computational work of a loop over several processors *without requiring modifications to the Fortran source program*.

Choice of which loops to parallelize and how they should be distributed can be left entirely up to the compiler (`-autopar`), determined explicitly by the programmer with source code directives (`-explicitpar`), or done in combination (`-parallel`).

Note – Programs that do their own (explicit) thread management should *not* be compiled with any of the compiler's parallelization options. Explicit multithreading (calls to `libthread` primitives) cannot be combined with routines compiled with these parallelization options.

Not all loops in a program can be profitably parallelized. Loops containing only a small amount of computational work (compared to the overhead spent starting and synchronizing parallel tasks) may actually run slower when parallelized. Also, some loops cannot be safely parallelized at all; they would compute different results when run in parallel due to dependencies between statements or iterations.

Sun compilers can detect loops that may be safely and profitably parallelized automatically. However, in most cases, the analysis is necessarily conservative, due to the concern for possible hidden side effects. (A display of which loops were and were not parallelized can be produced by the `-loopinfo` option.) By inserting source code directives before loops, you can explicitly influence the analysis, controlling how a specific loop is (or is not) to be parallelized. However, it then becomes your responsibility to ensure that such explicit parallelization of a loop does not lead to incorrect results.

Speedups—What to Expect

If you parallelize a program so that it runs over four processors, can you expect it to take (roughly) one fourth the time than it did with a single processor (a fourfold *speedup*)?

Probably not. It can be shown (by Amdahl's law) that the overall speedup of a program is strictly limited by the fraction of the execution time spent in code running in parallel. This is true *no matter how many processors are applied*. In fact, if *c* is the percentage of the execution time run in parallel, the theoretical speedup limit is 100/(100–*c*); therefore, if only 60% of a program runs in parallel, the *maximum* increase in speed is 2.5, independent of the number of processors. And with just four processors, the theoretical speedup for this program (assuming maximum efficiency) would be just 1.8 and not 4. With overhead, the actual speedup would be less.

As with any optimization, choice of loops is critical. Parallelizing loops that participate only minimally in the total program execution time has only minimal effect. To be effective, the loops that consume the *major* part of the run time *must* be parallelized. The first step, therefore, is to determine which loops are significant and to start from there.

Problem size also plays an important role in determining the fraction of the program running in parallel and consequently the speedup. Increasing the problem size increases the amount of work done in loops. A triply nested loop could see a cubic increase in work. If the outer loop in the nest is parallelized, a small increase in problem size could contribute to a significant performance improvement (compared to the unparallelized performance).

Steps to Parallelizing a Program

Here is a very general outline of the steps to parallelize an application:

1. *Optimize*. Use the appropriate set of compiler options to get the best serial performance on a single processor.
2. *Profile*. Using typical test data, determine the performance profile of the program. Identify the most significant loops.
3. *Benchmark*. Determine that the serial test results are accurate. Use these results and the performance profile as the benchmark.
4. *Parallelize*. Use a combination of options and directives to compile and build a parallelized executable.
5. *Verify*. Run the parallelized program on a single processor and check results to find instabilities and programming errors that might have crept in.
6. *Test*. Make various runs on several processors to check results.

7. *Benchmark*. Make performance measurements with various numbers of processors on a dedicated system. Measure performance changes with changes in problem size (scalability).

8. *Repeat* steps 4 to 7. Make improvements to parallelization scheme based on performance.

Data Dependency Issues

Not all loops are parallelizable. Running a loop in parallel over a number of processors may result in iterations out of order. Or, the multiple processors executing the loop in parallel may interfere with each other. These situations arise whenever there are data dependencies in the loop.

Recurrence

Variables that are set in one iteration of a loop and used in a subsequent iteration introduce cross-iteration dependencies, or *recurrences*. Recurrence in a loop requires that the iterations to be executed in the proper order. For example:

```
DO I=2,N
   A(I) = A(I-1)*B(I)+C(I)
END DO
```

requires the value computed for `A(I)` in the previous iteration to be used (as `A(I-1))` in the current iteration. To produce results running each iteration in parallel that are the same as with single processor, iteration `I` must complete before iteration `I+1` can execute.

Reduction

Reduction operations reduce the elements of an array into a single value. For example, summing the elements of an array into a single variable involves updating that variable in each iteration:

```
DO K = 1,N
  SUM = SUM + A(I)*B(I)
END DO
```

If each processor running this loop in parallel takes some subset of the iterations, the processors will interfere with each other, overwriting the value in `SUM`. For this to work, each processor must execute the summation one at a time, although the order is not significant.

Certain common reduction operations are recognized and handled as special cases by the compiler.

Indirect Addressing

Loop dependencies can result from stores into arrays that are indexed in the loop by subscripts whose values are not known. For example, indirect addressing could be order dependent if there are repeated values in the index array:

```
DO L = 1,NW
  A(ID(L)) = A(L) + B(L)
END DO
```

In the preceding, repeated values in `ID` cause elements in `A` to be overwritten. In the serial case, the last store is the final value. In the parallel case, the order is not determined. The values of `A(L)` that are used, old or updated, are order dependent.

Data Dependent Loops

It may be possible to rewrite a loop to eliminate data dependencies, making it parallelizable. However, extensive restructuring may be required.

Some general rules are:

- A loop is data *independent* only if all iterations write to distinct memory locations.
- Iterations may read from the same locations as long as no one iteration writes to them.

These are general conditions for parallelization. The compilers' automatic parallelization analysis considers additional criteria when deciding whether to parallelize a loop. However, the you can use directives to explicitly force loops to be parallelized, even loops that contain inhibitors and produce incorrect results.

Parallel Options and Directives Summary

The tables that follow list the `f77` 4.2 and `f90` 1.2 compilation options and directives related to parallelization.

Table 10-1 Parallelization Options for `f77`

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`
	Allocate local variables on stack	`-stackvar`
	Use Sun-style MP directives	`-mp=sun`
	Use Cray-style MP directives	`-mp=cray`

Table 10-2 Parallelization Options for `f90`

f90	Options	Syntax
	Explicit (*only*)	`-explicitpar`
	Automatic and Explicit	`-parallel`
	Automatic and Reduction and Explicit	`-parallel -reduction`
	Allocate local variables on stack	`-stackvar`

The following tables list `f77` and `f90` parallel directives.

Table 10-3 Parallel Directives for `f77`

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

Table 10-4 Parallel Directives for `f90`

f90	Parallel Directives	Purpose
	`!MIC$ DOALL` *optional qualifiers*	Parallelize next loop, if possible

Notes on Compiler Options

- `-reduction` requires `-autopar`.
- `-autopar` includes `-depend` and loop structure optimization.
- `-parallel` is equivalent to `-autopar -explicitpar`.
- `-noautopar, -noexplicitpar, -noreduction` are the negations.
- Parallelization options can be in any order, but they must be all lowercase.
- Reduction operations are not analyzed for explicitly parallelized loops.
- Use of any of the parallelization options requires a WorkShop license.

Specifying the Number of Processors

The environment variable PARALLEL controls the maximum number of processors available to the program:

```
demo% setenv PARALLEL 4
```

enables, for example, the execution of a program using at most four threads. If the target machine has four processors available, the threads will map to independent processors. If there are fewer than four processors available, some threads may run on the same processor as others, possibly degrading performance.

The Solaris command psrinfo(1M) displays a list of the processors available on a system:

```
demo% psrinfo
0   on-line   since 03/18/96 15:51:03
1   on-line   since 03/18/96 15:51:03
2   on-line   since 03/18/96 15:51:03
3   on-line   since 03/18/96 15:51:03
```

Stacks, Stack Sizes, and Parallelization

The executing program maintains a main memory stack for the parent program and distinct stacks for each thread. Stacks are temporary memory address spaces used to hold arguments and AUTOMATIC variables over subprogram invocations.

The default size of the main stack is about 8 megabytes. The Fortran compilers normally allocate local variables and arrays as STATIC (not on the stack). However, the -stackvar option forces allocation of *all* local variables and arrays on the stack (as if they were AUTOMATIC variables). Use of -stackvar is recommended with parallelization because it improves the optimizer's ability to parallelize CALLs in loops. -stackvar is *required* with explicitly parallelized loops containing subprogram calls. (See discussion of -stackvar in the *Fortran User's Guide.*)

The `limit` command displays the current main stack size as well as setting it:

```
demo% limit
cputime unlimited
filesize unlimited
datasize 2097148 kbytes
stacksize 8192 kbytes              <- current main stack size
coredumpsize 0 kbytes
descriptors 64
memorysize unlimited
demo% limit stacksize 65536        <- set main stack to 64Mb
```

Each thread of a multithreaded program has its own *thread* stack. This stack mimics the main program stack but is unique to the thread. The thread's `PRIVATE` arrays and variables (local to the thread) are allocated on the thread stack. The default size is 256 kilobytes. The size is set with the `STACKSIZE` environment variable:

```
demo% setenv STACKSIZE 8192      <- Set thread stack size to 8 Mb
```

Setting the thread stack size to a value larger than the default may be necessary for most parallelized Fortran codes. However, it may not be possible to know just how large to set it, except by trial and error, especially if private/local arrays are involved. If the stack size is too small for a thread to run, the program will abort with a segmentation fault.

10

Automatic Parallelization

With the `f77` option `-autopar` and the `f90` option `-parallel`, the compilers automatically find those `DO` loops that can be parallelized effectively. These loops are then transformed to distribute their iterations evenly over the available processors. The compiler generates the threads calls needed in the compiled code to make this happen.

Loop Parallelization

The compiler's dependency analysis transforms a `DO` loop into a parallelizable task. The compiler may restructure the loop to split out unparallelizable sections that will run serially. It then distributes the work evenly over the available processors. Each processor executes a different chunk of iterations.

Example: With four CPUs and a parallelized loop with 1000 iterations:

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

Only loops that do not depend on the order in which the computations are performed can be successfully parallelized. The compiler's dependency analysis rejects loops with inherent data dependencies. If it cannot fully determine the data flow in a loop, the compiler acts conservatively and does not parallelize. Also, it may choose not to parallelize a loop if it determines the performance gain does not justify the overhead.

Note that the compiler always chooses to parallelize loops using a *chunk* distribution—simply dividing the work in the loop into equal blocks of iterations. Other distribution schemes may be specified using explicit parallelization directives described later in this chapter.

Definitions: Array, Scalar, and Pure Scalar

A few definitions, from the point of view of *automatic parallelization*, are needed:

An *array* is a variable that is declared with at least one dimension.

A *scalar* is a variable that is not an array.

A *pure scalar* is a scalar variable that is not aliased—not referenced in an `EQUIVALENCE` or `POINTER` statement.

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* scalars.

Automatic Parallelization Criteria

`DO` loops that have no cross-iteration data dependencies are automatically parallelized by `-autopar` (`f77`) or `-parallel` (`f90`). The general criteria for automatic parallelization are:

- `DO` loops are parallelized, but not `DO WHILE`.
- The values of *array* variables for each iteration of the loop must not depend on the values of *array* variables for any other iteration of the loop.
- Calculations within the loop must not *conditionally* change any pure scalar variable that is referenced after the loop terminates.
- Calculations within the loop must not change a *scalar* variable across iterations. This is called a *loop-carried dependency.*

Apparent Dependencies (`f77` only)

The compiler may automatically eliminate a reference that appears to create a dependency transforming the compiled code. One of the many such transformations makes use of private versions of some of the arrays. Typically, the compiler does this if it can determine that such arrays are used in the original loops only as temporary storage.

Example: Using `-autopar`, with dependencies eliminated by private arrays:

```
parameter (n=1000)
real a(n), b(n), c(n,n)
do i = 1, 1000            <--Parallelized
  do k = 1, n
    a(k) = b(k) + 2.0
  end do
  do j = 1, n
    c(i,j) = a(j) + 2.3
  end do
end do
end
```

In the preceding example, the outer loop is parallelized and run on independent processors. Although the inner loop references to array `a(*)` appear to result in a data dependency, the compiler generates temporary private copies of the array to make the outer loop iterations independent.

Inhibitors to Automatic Parallelization

Under automatic parallelization, the compilers do not parallelize a loop if:

- The `DO` loop is nested inside another `DO` loop that is parallelized.
- Flow control allows jumping out of the `DO` loop.
- A user-level subprogram is invoked inside the loop.
- An I/O statement is in the loop.
- Calculations within the loop change an aliased scalar variable.

The following additional inhibitors exist for the `f90` 1.2 compiler:

- The step size of the `DO` loop is a variable.
- The `DO` loop is the innermost in a nest or is a singly-nested loop.

Nested Loops

On multiprocessor systems, it is most effective to parallelize the outermost loop in a loop nest, rather than the innermost. Because parallel processing typically involves relatively large loop overhead, parallelizing the outermost loop minimizes the overhead and maximizes the work done for each processor. Under automatic parallelization, the compilers start their loop analysis from

the outermost loop in a nest and work inward until a parallelizable loop is found. Once a loop within the nest is parallelized, loops contained within the parallel loop are passed over.

Note – `f90` 1.2: Innermost or singly-nested loops are not automatically parallelized.

Automatic Parallelization With Reduction Operations

A computation that transforms an array into a scalar is called a *reduction operation*. Typical reduction operations are the sum or product of the elements of a vector. Reduction operations violate the criterion that calculations within a loop not change a scalar variable in a cumulative way across iterations.

Example: Reduction summation of the elements of a vector:

```
s = 0.0
do i = 1, 1000
  s = s + v(i)
end do
t(k) = s
```

However, for some operations, if the reduction is the only factor that prevents parallelization, it is still possible to parallelize the loop. Common reduction operations occur so frequently that the compilers are capable of recognizing and parallelizing them as special cases.

Recognition of reduction operations is not included in the automatic parallelization analysis unless the `-reduction` compiler option is specified along with `-autopar` or `-parallel`.

If a parallelizable loop contains one of the reduction operations listed in Table 10-5, the compiler will parallelize it if `-reduction` is specified.

Recognized Reduction Operations

The following table lists the reduction operations that are recognized by `f77`.

Table 10-5 Recognized Reduction Operations (`f77`)

Mathematical Operations	**Fortran Statement Templates**
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))` *(See Note below)*
Maximum of the elements	`s = amax( s, v(i))` *(See Note below)*

`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 – All forms of the `MIN` and `MAX` functions are recognized.

Numerical Accuracy and Reduction Operations

Floating-point sum or product reduction operations may be inaccurate due to the following conditions:

- The order in which the calculations were performed in parallel was not the same as when performed serially on a single processor.
- The order of calculation affected the sum or product of floating-point numbers. Hardware floating-point addition and multiplication are not associative. Roundoff, overflow, or underflow errors may result depending on how the operands associate. For example, `(X*Y)*Z` and `X*(Y*Z)` may not have the same numerical significance.

In some situations, the error may not be acceptable.

Example: 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, roundoff error on the order of 10^{-14} is acceptable for data that is random to begin with. For more information, see the Sun *Numerical Computation Guide.*

Explicit Parallelization

This section describes the source code directives recognized by `f77` 4.2 and `f90` 1.2 to explicitly indicate which loops to parallelize and what strategy to use.

Explicit parallelization of a program requires prior analysis and deep understanding of the application code as well as the concepts of shared-memory parallelization.

Note – Be aware that there are differences in directive syntax and features between the `f77` and `f90` implementations. `f77` accepts *either* Sun *or* Cray style directives, while `f90` accepts *only* Cray style directives.

`DO` loops are marked for parallelization by directives placed immediately before them. The compiler options `-parallel` and `-explicitpar` must be used for `DO` loops to be recognized and parallel code generated. Take care when choosing which loops to mark for parallelization. The compiler generates threaded, parallel code for all loops marked with `DOALL` directives, even if there are data dependencies that will cause the loop to compute incorrect results when run in parallel.

If you do your own multithreaded coding using the `libthread` primitives, do *not* use any of the compilers' parallelization options—the compilers cannot parallelize code that has already been parallelized with user calls to the threads library.

Parallelizable Loops

A loop is appropriate for explicit parallelization if:

- It is a `DO` loop, 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, 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: Private and Shared

A *private* variable or array is private to a *single iteration* of a loop. The value assigned to a private variable or array in one iteration is not propagated to any other iteration of the loop.

A *shared* variable or array is shared with all other iterations. The value assigned to a shared variable or array in an iteration is seen by other iterations of the loop.

If an explicitly parallelized loop contains shared references, then you must ensure that sharing does not cause correctness problems. The compiler does no synchronization on updates or accesses to shared variables.

If you specify a variable as private in one loop, and its only initialization is within some other loop, the value of that variable may be left undefined in the loop.

Default Scoping Rules for Sun-Style Directives

For Sun-style (`C$PAR`) explicit directives, the compiler uses default rules to determine whether a scalar or array is shared or private. You can override the default rules to specify the attributes of scalars or arrays referenced inside a loop. (With Cray-style `!MIC$` directives, all variables that appear in the loop must be explicitly declared either shared or private on the `DOALL` directive.)

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 preceeding 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 fix the example by using `C$PAR DOALL PRIVATE(y)`.

Sun-Style Parallelization Directives (`f77` only)

Parallelization directives are comment lines that tell the compiler to parallelize (or not to parallelize) the `DO` loop that follows the directive. Directives are also called *pragmas*.

A parallelization directive consists of one or more *directive lines*.

Sun-style directives are recognized by `f77` by default (or with the `-mp=sun` option). Cray-style or `f90` directives are discussed on page 167. A Sun-style directive line is defined as follows:

```
C$PAR  Directive    [ Qualifiers ]      <- Initial directive line
C$PAR&  [More_Qualifiers]               <- Optional continuation lines
```

- The letters of a directive line case-insensitive.
- The first five 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 unless the `-e` option is specified.

The parallel directives and their actions are as follows:

Directive	Action
DOALL	Parallelize the next loop.
DOSERIAL	Do not parallelize the next loop.
DOSERIAL*	Do not parallelize the next nest of loops.

Examples: f77 parallel directives:

```
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 *Directive*

The compilers will parallelize the DO loop following a DOALL directive (if compiled with the -parallel or -explicitpar options).

Note – Analysis and transformation of reduction operations within loops is not done if they are explicitly parallelized.

Example: Explicit parallelization of a loop:

```
demo% cat t4.f
    ...
C$PAR DOALL
    do i = 1, n
       a(i) = b(i) * c(i)
    end do
    do k = 1, m
       x(k) = x(k) * z(k,k)
    end do
    ...
demo% f77 -explicitpar t4.f
```

CALL *in a Loop*

A subprogram call in a loop (or in any subprograms called from within the called routine) may introduce data dependencies that could go unnoticed without a deep analysis of the data and control flow through the chain of calls. While it is best to parallelize outermost loops that do a significant amount of the work, these tend to be the very loops that involve subprogram calls.

Because such an interprocedural analysis is difficult and could greatly increase compilation time, automatic parallelization modes do not attempt it. With explicit parallelization, the compiler generates parallelized code for a loop marked with a `DOALL` directive that contains calls to subprograms. It is still the programmer's responsibility to insure that no data dependencies exist within the loop and all that the loop encloses, including called subprograms.

Multiple invocations of a routine from different processors may cause problems resulting from references to local static variables that interfere with each other. Making all the local variables in a routine *automatic* rather than *static* will prevent this problem. Each invocation of a subprogram will then have its own unique store of local variables maintained on the stack, and no two invocations will interfere with each other.

Local subprogram variables can be made automatic variables that reside on the stack either by listing them on an `AUTOMATIC` statement or by compiling the subprogram with the `-stackvar` option. However, local variables initialized in `DATA` statements must be rewritten to be initialized in actual assignments.

Note – Allocating local variables to the stack may cause stack overflow. See page 140 about increasing the size of the stack.

Data dependencies can still be introduced through the data passed down the call tree as arguments or through COMMON blocks. This data flow should be analyzed carefully before parallelizing a loop with subprogram calls.

DOALL *Qualifiers*

All qualifiers on the DOALL directive are optional. Table 10-6 summarizes them.

Table 10-6 DOALL Qualifiers

Qualifiers	Action	Syntax
PRIVATE	Do not share variables *u1*, … 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 last DO iteration values of all *private* variables.	DOALL SAVELAST
STOREBACK	Save the last DO iteration values of variables *v1*, …	DOALL STOREBACK(*v1*,*v2*,…)
REDUCTION	Treat the variables *v1*, *v2*, … as *reduction* variables.	DOALL REDUCTION(*v1*,*v2*,…)
SCHEDTYPE	Set the scheduling type to *t*.	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 thread 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, 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 preceding 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, 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 preceding 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 does not need use a separate copy of that variable or array for each thread 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 preceding example, x is a shared variable but the compiler can rely on the fact that it will not change over each iteration of the i loop because of its READONLY specification.

STOREBACK(*varlist*)

A STOREBACK variable or array is one whose value is computed in a DOALL loop. The 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 preceding 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, even if this last iteration is the same 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 preceding 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) and performs the STOREBACK operation for x, may not be the same processor that currently contains the last updated value of x. The compiler issues a warning message about 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; 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 preceding 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 if a DOALL loop 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 preceeding 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.

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.
FACTORING[(*m*)]	Use *factoring* scheduling for this DO loop. With *n* iterations initially and *k* CPUs, distribute *n/(2k)* iterations uniformly to each processor until all iterations have been processed. • At least *m* iterations must be assigned to each processor. • 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, and so on.
GSS[(*m*)]	Use *guided self-scheduling* for this DO loop. With *n* iterations initially, and *k* CPUs, then: • Assign *n/k* iterations to the first processor. • Assign the remaining iterations divided by *k* to the second processor, and so on until all iterations have been processed. Note: • At least *m* iterations must be assigned to each CPU. • 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, and so on.

Multiple Qualifiers

Qualifiers can appear multiple times with cumulative effect. In the case of conflicting qualifiers, the compiler issues a warning message, and the qualifier appearing last prevails.

Example: A three-line Sun-style 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 preceding 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. This directive applies to the one loop immediately following it (if you compile it 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 preceding 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. This directive applies to the whole nest of loops immediately following it (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 preceeding 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 preceeding 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. The directive 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 preceeding example, DOSERIAL* applies only to the i loop and not to the j loop, regardless of whether the call to the subroutine callee is inlined.

Inhibitors to Explicit Parallelization

In general, the compiler parallelizes a loop if you explicitly direct it to. There are exceptions—some loops the compiler just cannot parallelize.

The following are the primary detectable inhibitors that may prevent explicitly parallelizing a DO loop.

- The DO loop is nested inside another DO loop that is parallelized.

  This exception holds for indirect nesting, too. If you explicitly parallelize a loop that includes a call to a subroutine, then even if you parallelize loops in that subroutine, those loops are not run in parallel at runtime.

- A flow control statement allows jumping out of the DO loop.
- The index variable of the loop is subject to side effects, such as being equivalenced.

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. The following list of typical parallelization problems shows those that are ignored by the compiler and those that generate messages with `-vpara`.

Table 10-7 Explicit Parallelization Problems

Problem	**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 not predictable.*	Yes	No

Example: Nested loops:

```
    ...
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 parallelized loop in subroutine:

```
C$PAR DOALL
    do 100 i = 1, 200
      ...
      call calc (a, x)
      ...
100 continue
    ...
demo% f77 -explicitpar -vpara t.f
```

At runtime, the loop may run in parallel.

```
    subroutine calc ( b, y )
    ...
C$PAR DOALL
    do 1 m = 1, 1000
      ...
1   continue
    return
    end
```

At runtime, both loops do not run in parallel.

In the preceeding example, the loop within the subroutine is not parallelized because the subroutine itself is run in parallel.

Example: Jumping out of loop:

```
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:

```
    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:

```
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.

Example: I/O statement in loop

```
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:

```
    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 preceeding example, the program may deadlock in `libF77_mt` and hang. Type Control-C to regain keyboard control.

There are situations where the programmer may not be aware that I/O could take place within a parallelized loop. Consider a user-supplied exception handler that prints output when it catches an arithmetic exception (like divide by zero). If a parallelized loop provokes an exception, the implicit I/O from the handler may cause I/O deadlocks and a system hang.

In general:

- The library `libF77_mt` is MT safe, but mostly not MT hot.
- You cannot 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. 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 last, wins!).

An interface is colloquially called *MT hot* if the implementation has been tuned for performance advantage, using the techniques of multithreading. For some formal definitions of multithreading technology, read *The Solaris Multithreaded Programming Guide*. See also the Threads page by searching for "threads" at: `http://www.sun.com/search/search.html`

Cray-Style Parallelization Directives

Parallel directives have two forms: Sun style and Cray style. The `f77` default is Sun style (`-mp=sun`). To use Cray-style directives with `f77`, you must compile with `-mp=cray`. Only Cray-style directives are available with `f90`.

Mixing program units compiled with both Sun and Cray directives can produce different results.

A major difference between Sun and Cray directives is that Cray style *requires explicit scoping of every scalar and array in the loop* as either `SHARED` or `PRIVATE`.

The following table shows Cray style directive syntax.

Table 10-8 Overview of Alternate Directive Syntax

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 directive line is case insensitive.
- The first five 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.

With `f90 -free` free-format, leading blanks may appear before `!MIC$`.

Qualifiers (Cray Style)

For Cray-style directives, the `PRIVATE` qualifier is required. Each variable within the `DO` loop must be qualified as private or shared, and the `DO` loop index must always be private. Table 10-9 summarizes available Cray-style qualifiers.

Table 10-9 `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 single scheduling qualifier, for example, !MIC$& CHUNKSIZE(100). Table 10-10 shows the Cray-style DOALL directive scheduling qualifiers:

Table 10-10 DOALL Cray Scheduling

Qualifier	Action
GUIDED	Distribute the iterations by use of guided self-scheduling. This distribution minimizes synchronization overhead, with acceptable dynamic load balancing.
SINGLE	Distribute *one* iteration to each available processor.
CHUNKSIZE(*n*)	Distribute *n* iterations to each available processor. *n* may be 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 *n* iterations, then distribute *n/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.

The f77 default scheduling type is the Sun-style STATIC. The f90 default is GUIDED.

Inhibitors to f90 *Explicit Parallelization*

In addition to the explicit parallelization problems listed on page 162, the parallelization inhibitors for f90 include:

- The DO increment parameter, if specified, is a variable.
- There is an I/O statement in the loop.
- Parallelized loops in subprograms called from parallelized loops are, in fact, not run in parallel.

Debugging Parallelized Programs

Compiling with the -g option cancels any of the parallelization options -autopar, -explicitpar, and -parallel, as well as -reduction and -depend. Some alternative ways to debug parallelized code are suggested in the following section.

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—Verify that the program works correctly by compiling with `-O3` or `-O4`, but without any parallelization.
  - 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.

  Check also for out of bounds array references by compiling with `-C`.

  Problems using `-autopar` may indicate that the compiler is parallelizing something it should not.

- Turn off `-reduction`.

  If you are using the `-reduction` option, summation reduction may be occurring and yielding slightly different answers. Try running without this option.

- Reduce the number of compile options.

  Compile with just `-parallel -O3` and check the results.

- 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 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 that are being parallelized. Recompile and execute. Use `-loopinfo` to see which loops are being parallelized.

  Continue this process until you start getting the correct results.

  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 that are being parallelized. Compile with `-explicitpar`, then execute and verify the results. Use `-loopinfo` to see which loops 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 *backward* serially.

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

- 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.

Replace:

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

With:

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

One Possible Way 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 to loop1.f
Part 2 to 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.

```
demo% f77 -O3 -c -explicitpar loop1.f
```

Compile Part 2: dbx, no parallel.

```
demo% f77 -c -g loop2.f
```

Link both into a.out.

```
demo% f77 loop1.o loop2.o -explicitpar
```

Start a.out under dbx control.
Put a breakpoint into the loop body and run.

```
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)
```

dbx stops at the breakpoint.

```
t@1 (l@1) stopped in loop_body at line 2 in file "loop2.f"
    2           write(0,*) 'Iteration ', k
```

Show k.
See the dbx documentation.

```
(dbx) print k
k = 1                      ← Various values other than 1 are possible
(dbx)
```

10

C-Fortran Interface 11

This chapter treats issues regarding Fortran and C interoperability.

The discussion is inherently limited to the specifics of the Sun Fortran 77, Fortran 90, and C compilers.

Note – Material common to both Sun Fortran 77 and Fortran 90 is presented in examples that use Fortran 77.

Compatibility Issues

Most C-Fortran interfaces must agree 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 libraries

Some C-Fortran interfaces must also agree on:

- Arrays: indexing and order
- File descriptors and `stdio`
- File permissions

Function or Subroutine

The word *function* has different meanings in C and Fortran:

- In C, all subprograms are functions; however, some may return a null (`void`) value.
- In Fortran, a function passes a return value, but a subroutine does not.

Fortran Calls a C Function

- If the called C function returns a value, call it from Fortran as a function.
- If the called C function does not return a value, call it as a subroutine.

C Calls a Fortran Subprogram

- If the called Fortran subprogram is a *function*, call it from C as a function that returns a compatible data type.
- If the called Fortran subprogram is a *subroutine*, call it from C as a function that returns a value of `int` (compatible to Fortran `INTEGER*4`) or `void`. A value is returned if the Fortran subroutine uses alternate returns, in which case it is the value of the expression on the `RETURN` statement. If no expression appears on the `RETURN` statement, zero is returned.

Data Type Compatibility

Fortran 77 vs. C Data Types

Table 11-1 shows the sizes and allowable alignments for Fortran 77 data types. It assumes no compilation options effecting alignment or promoting default data sizes are applied.

Table 11-1 Data Sizes and Alignments—Pass by Reference (`f77` vs. `cc`)

Fortran 77 Data Type	C Data Type	Size (Bytes)	Alignment (Bytes) SPARC	Intel	PowerPC
`BYTE X`	`char x`	1	1	1	1
`CHARACTER X`	`char x`	1	1	1	1
`CHARACTER*`*n* `X`	`char x[n]`	n	1	1	1
`COMPLEX X`	`struct {float r,i;} x;`	8	4	2/4	4
`COMPLEX*8 X`	`struct {float r,i;} x;`	8	4	2/4	4
`DOUBLE COMPLEX X`	`struct {double dr,di;}x;`	16	4/8	2/4	8
`COMPLEX*16 X`	`struct {double dr,di;}x;`	16	4/8	2/4	8
`COMPLEX*32 X`	`struct {long double dr,di;} x;`	32	4/8	—	8
`DOUBLE PRECISION X`	`double x`	8	4/8	2/4	8
`REAL X`	`float x`	4	4	2/4	4
`REAL*4 X`	`float x`	4	4	2/4	4
`REAL*8 X`	`double x`	8	4/8	2/4	8
`REAL*16 X`	`long double x`	16	4/8	—	8
`INTEGER X`	`int x`	4	4	2/4	4
`INTEGER*2 X`	`short x`	2	2	2	2
`INTEGER*4 X`	`int x`	4	4	2/4	4
`INTEGER*8 X`	`long long int x`	8	4	2/4	8
`LOGICAL X`	`int x`	4	4	2/4	4
`LOGICAL*1 X`	`char x`	1	1	1	1
`LOGICAL*2 X`	`short x`	2	2	2	2
`LOGICAL*4 X`	`int x`	4	4	2/4	4
`LOGICAL*8 X`	`long long int x`	8	4	2/4	8

Note the following:

- C data types `int`, `int long`, and `long`, are equivalent (4 bytes).
- `REAL*16` and `COMPLEX*32` are only available on SPARC and PowerPC.

- 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 marked 2/4 for Intel indicate that either two byte or four byte alignment is possible, but two byte can result in a performance degradation.
- Alignments marked 4/8 for SPARC indicate that either four byte or eight byte alignment is possible, but four byte can result in a performance degradation.
- Alignments shown are for f77 data types.
- The elements and fields of arrays and structures must be compatible.
- You cannot pass arrays, character strings, or structures by value.
- You can pass arguments by value from f77 to C, but not from C to f77, since the %VAL() does not work in a SUBROUTINE statement.

Fortran 90 vs. C Data Types

The following table similarly compares the Fortran 90 data types with C:

Table 11-2 Data Sizes and Alignment—Pass by Reference (f90 vs. cc) (*SPARC only*)

Fortran 90 Data Type	C Data Type	Size (Bytes)	Alignment (Bytes)
CHARACTER x	unsigned char x ;	1	1
CHARACTER (LEN=n) x	unsigned char x[*n*] ;	*n*	1
CHARACTER (LEN=n, KIND=1) x	unsigned char x[*n*] ;	*n*	1
COMPLEX x	struct {float r,i;} x;	8	4
COMPLEX (KIND=4) x	struct {float r,i;} x;	8	4
COMPLEX (KIND=8) x	struct {double dr,di;} x;	16	4
DOUBLE PRECISION x	double x ;	8	4
REAL x	float x ;	4	4
REAL (KIND=4) x	float x ;	4	4
REAL (KIND=8) x	double x ;	8	4
INTEGER x	int x ;	4	4
INTEGER (KIND=1) x	signed char x ; *See Note*	(1)	4
INTEGER (KIND=2) x	short x ; *See Note*	(2)	4
INTEGER (KIND=4) x	int x ;	4	4
LOGICAL x	int x ;	4	4
LOGICAL (KIND=1) x	signed char x ;	1	4
LOGICAL (KIND=2) x	short x ;	2	4
LOGICAL (KIND=4) x	int x ;	4	4

Note the following:

- In this release (`f90` 1.2), `INTEGER`, for `KIND=1`, `2`, or `4`, take 4 bytes, align on 4-byte boundaries, and use 32-bit arithmetic.
- C data types `int`, `int long`, and `long`, are equivalent (4 bytes).

Case Sensitivity

C and Fortran take opposite perspectives on case sensitivity:

- C is case sensitive—uppercase or lowercase matters.
- Fortran ignores case.

The `f77` and `f90` 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 on function/subprogram names.

Use one of these two solutions, but not both.

Most examples in this chapter use all lowercase letters for the name in the C function, and do *not* use the `f77 -U` compiler option. (`f90` 1.2 does not have an equivalent option.)

Underscore in Names of Routines

The Fortran compiler normally appends an underscore (_) to the names of subprograms appearing both at entry point definition and in calls. This convention differs from C procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All Fortran library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are three usual solutions to the underscore problem:

- In the C function, change the name of the function by appending an underscore to that name.

- Use the `f77 C()` pragma to tell the Fortran 77 compiler to omit those trailing underscores.
- Use the `f77 -ext_names` option to make external names without underscores.

Use only one of these solutions.

The examples in this chapter could use the Fortran 77 `C()` compiler pragma to avoid 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 compiler does not append an underscore 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, the C function does not need an underscore appended to the function name.

This release of Fortran 90 (1.2) does not have equivalent methods for avoiding underscores. Trailing underscores are required in the names of C routines called from Fortran 90 routines.

Argument-Passing by Reference or Value

In general, Fortran routines pass arguments by reference. In a call, if you enclose an argument with the `f77` nonstandard function `%VAL()`, the calling routine passes it by value.

In general, C passes arguments by value. If you precede an argument by the ampersand operator (&), C passes the argument by reference using a pointer. C always passes arrays and character strings by reference.

Argument Order

Except for arguments that are character strings, Fortran and C pass arguments in the same order. However, for every argument of character type, the Fortran routine passes an additional argument giving the length of the string. These are `long int` quantities in C, passed by value.

The order of arguments is:

- Address for each argument (datum or function)
- A `long int` for each character argument (the whole list of string lengths comes after the whole list of other arguments).

Example:

This Fortran code fragment:	**Is equivalent to this in C:**
`CHARACTER*7 S`	`char s[7];`
`INTEGER B(3)`	`long b[3];`
…	…
`CALL SAM( S, B(2) )`	`sam_( s, &b[1], 7L ) ;`

Array Indexing and Order

Array indexing and order differ between Fortran and C.

Array Indexing

C arrays always start at zero, but by default Fortran arrays start at 1. There are two usual ways of approaching indexing.

- You can use the Fortran default, as in the preceeding example. Then the Fortran element `B(2)` is equivalent to the C element `b[1]`.
- You can specify that the Fortran array `B` starts at `B(0)` as follows:

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

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

Array Order

Fortran arrays are stored in column-major order: `A(3,2)`

`A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2) A(1,3) A(2,3) A(3,3)`

C arrays in row-major order: `A[3][2]`

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

For one-dimensional arrays, this is no problem. For two-dimensional and higher arrays, be aware of how subscripts appear and are used in all references and declarations—some adjustments may be necessary.

For example, it may be confusing to do part of a matrix manipulation in C and the rest in Fortran. It may be preferable to pass an *entire* array to a routine in the other language and perform *all* the matrix manipulation in that routine to avoid doing part in C and part in Fortran.

File Descriptors and `stdio`

Fortran 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 runtime system translates from one to the other, so most Fortran programs do not have to recognize file descriptors.

Many C programs use a set of subroutines, called *standard I/O* (or `stdio`). Many functions of Fortran I/O use standard I/O, which in turn uses operating system I/O calls. Some of the characteristics of these I/O systems are listed in Table 11-3.

Table 11-3 Comparing Fortran and C I/O

	Fortran 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	Byte stream	Byte stream
Form	Arbitrary nonnegative integers	Pointers to structures in the user's address space	Integers from 0-63

File Permissions

C programmers typically open input files for reading and output files for writing or for reading and writing. In Fortran, it is not possible for the system to foresee what use you will make of a file, since there is no parameter to the `OPEN` statement that gives that information.

Fortran 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.

Libraries and Linking With the `f77` or `f90` Command

To link the proper Fortran and C libraries, use the `f77` or `f90` command to invoke the linker.

Example 1: Use `f77` to link:

```
demo% cc -c RetCmplxmain.c
demo% f77 RetCmplx.f RetCmplxmain.o ← This command line does the linking.
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

Passing Data Arguments by Reference

The standard method for passing data between Fortran routines and C procedures is by reference. To a C procedure, a Fortran subroutine or function call looks like a procedure call with all arguments represented by pointers. The only peculiarity is the way Fortran handles character strings as arguments and as the return value from a `CHARACTER*n` function.

Simple Data Types

- For simple data types (not `COMPLEX` or `CHARACTER` strings), define or pass each associated argument in the C routine as a pointer:

Code Example 11-1 Passing Simple Data Types

Fortran calls C	C calls Fortran

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

COMPLEX *Data*

- Treat Fortran COMPLEX data as a pointer to a C struct of two floats or two doubles:

Code Example 11-2 Passing COMPLEX Data Types

Fortran calls C	**C calls Fortran**

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

Character Strings

Passing strings between C and Fortran routines is not recommended because there is no standard interface. However, note the following rules:

- All C strings are passed by reference.
- Fortran calls pass an additional argument for every character type in the argument list. The extra argument gives the length of the string and is equivalent to a C long int passed by value. (This is implementation dependent.) The extra string-length arguments appears after the explicit arguments in the call.

A Fortran call with a character string argument is shown below with its C equivalent:

Code Example 11-3 Passing a CHARACTER string

Fortran call:	**C equivalent:**

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

If the length of the string is not needed in the called routine, the extra arguments may be ignored. However, note that Fortran does not automatically terminate strings with the explicit null character that C expects. This must be added by the calling program.

One-Dimensional Arrays

- Array subscripts in C start with 0.

Code Example 11-4 Passing a One-Dimensional Array

Fortran calls C	**C calls Fortran**

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

Two-Dimensional Arrays

- Rows and columns between C and Fortran are switched.

Code Example 11-5 Passing a Two-Dimensional Array

Fortran calls C	**C calls Fortran**

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

Structures

- C and Fortran 77 structures and Fortran 90 derived types can be passed to each other's routines as long as the corresponding elements are compatible.

Code Example 11-6 Passing Fortran 77 STRUCTURE Records

Fortran calls C

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

C calls Fortran

```
struct point {
   float x,y,z;
};
void fflip_ ( struct point *) ;
  ...
  struct point d;
  struct point *ptx = &d;
  ...
  fflip_ (ptx);
  ...
-------------------------------
  SUBROUTINE FFLIP(P)
  STRUCTURE /POINT/
     REAL X,Y,Z
  END STRUCTURE
  RECORD /POINT/ P
  REAL T
  T = P.X
  P.X = P.Y
  P.Y = T
  P.Z = -2.*P.Z
  ...
```

Code Example 11-7 Passing Fortran 90 Derived Types

Fortran 90 calls C

```
  TYPE point
    REAL :: x, y, z
  END TYPE point
  TYPE (point) base
  EXTERNAL flip
  ...
  CALL flip( base)
  ...
-------------------------------
struct point {
   float x,y,z;
};
void flip_( v )
struct point *v;
{
      float t;
      t = v -> x;
      v -> x = v -> y;
      v -> y = t;
      v -> z = -2.*(v -> z);
}
```

C calls Fortran 90

```
struct point {
   float x,y,z;
};
extern void fflip_ (
              struct point *) ;
  ...
  struct point d;
  struct point *ptx = &d;
  ...
  fflip_ (ptx);
  ...
-------------------------------
  SUBROUTINE FFLIP( P )
    TYPE POINT
       REAL :: X, Y, Z
    END TYPE POINT
    TYPE (POINT) P
    REAL :: T
    T = P%X
    P%X = P%Y
    P%Y = T
    P%Z = -2.*P%Z
    ...
```

Pointers

- A Fortran 77 `pointer` can be passed to a C routine as a pointer to a pointer because the Fortran routine passes arguments by reference. A Fortran 77 pointer is not equivalent, however, to a C `char **` data type.

Code Example 11-8 Passing Fortran 77 `POINTER`

Fortran calls C	C calls Fortran

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

- C pointers are not compatible with Fortran 90.

Passing Data Arguments by Value

Call by value is only available for simple data with Fortran 77, and only by Fortran routines calling C routines. There is no way for a C routine to call a Fortran routine and pass arguments by value. It is not possible to pass arrays, character strings, or structures by value. These are best passed by reference.

- Use the nonstandard Fortran 77 function `%VAL(`*arg*`)` as an argument in the call.

In the example, the Fortran routine passes `x` by value and `y` by reference. The C routine incremente both `x` and `y`, but only `y` is changed:

Code Example 11-9 Passing Simple Data Arguments by Value: Fortran 77 Calling C

Fortran calls C

```
  REAL x, y
  x = 1.
  y = 0.
  PRINT *, x,y
  CALL value( %VAL(x), y)
  PRINT *, x,y
  END
-------------------------------------------------------
void value_( float x, float *y)
{
  printf("%f, %f\n",x,*y);
  x = x + 1.;
  y = y + 1.;
  printf("%f, %f\n",x,*y);
}
-------------------------------------------------------
```

Compiling and running produces output:

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

Functions that Return a Value

A Fortran function that returns a value of type `BYTE` (*Fortran 77 only*), `INTEGER`, `REAL`, `LOGICAL`, `DOUBLE PRECISION`, or `REAL*16` (*SPARC and PowerPC only*) is equivalent to a C function that returns a compatible type (see Table 11-1 and Table 11-2). There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

Returns Simple Data Type

- The following example returns a `REAL` or `float` value. `BYTE`, `INTEGER`, `LOGICAL`, `DOUBLE PRECISION`, and `REAL*16` are treated in a similar way:

Code Example 11-10 Functions Returning a Value – `REAL` and `float`

Fortran calls C	C calls Fortran

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

Returns `COMPLEX` *Data*

- A Fortran function returning `COMPLEX` or `DOUBLE COMPLEX` is equivalent to a C function with an additional first argument that points to the return value in memory. The general pattern for the Fortran function and its corresponding C function is:

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

This is shown in the following example:

Code Example 11-11 Function Returning `COMPLEX` *(Fortran 77 only)*

Fortran calls C	C calls Fortran

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

Fortran 90 `COMPLEX` function type is incompatible with this implementation.

Returns CHARACTER *String*

- Passing strings between C and Fortran routines is not encouraged. However, a Fortran character-string-valued function is equivalent to a C function with two additional first arguments—data address and string length. The general pattern for the Fortran function and its corresponding C function is:

Fortran function	C function

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

Here is an example:

Code Example 11-12 Function Returning CHARACTER String

Fortran calls C

```
  CHARACTER STRING*16, CSTR*9
  STRING = ' '
  STRING = '123' // CSTR('*',9)
  ...
-----------------------------
 void cstr_( char *p2rslt,
             int rslt_len,
             char *p2arg,
             int *p2n,
             int arg_len )
{ /* return n copies of arg */
    int count, i;
    char *cp;
    count = *p2n;
    cp = p2rslt;
    for (i=0; i<count; i++) {
       *cp++ = *p2arg ;
    }
}
```

C calls Fortran

```
 void fstr_( char *, int,
          char *, int *, int );
 char sbf[9] = "123456789";
 char *p2rslt = sbf;
 int rslt_len = sizeof(sbf);
 char ch = '*';
 int n = 4;
 int ch_len = sizeof(ch);
 /* make n copies of ch in sbf
  */
   fstr_( p2rslt, rslt_len,
          &ch, &n, ch_len );
  ...
-----------------------------
  FUNCTION FSTR( C, N)
  CHARACTER FSTR*(*), C
  FSTR = ''
  DO I = 1,N
    FSTR(I:I) = C
  END DO
  FSTR(N+1:N+1) = CHAR(0)
  END
```

In this example, the C function and calling C routine must accommodate two initial extra arguments (pointer to result string and length of string) and one additional argument at the end of the list (length of character argument). Note that in the Fortran routine called from C, it is necessary to explicitly add a final null character.

Labeled `COMMON`

Fortran labeled `COMMON` can be emulated in C by using a global `struct`:

Code Example 11-13 Labeled `COMMON`

Fortran COMMON Definition	**C "COMMON" Definition**
`COMMON /BLOCK/ ALPHA,NUM` `...`	see below

```
extern struct block {
    float alpha;
    int num;
    };
extern struct block block_ ;

main ()
{
  ...
  block_.alpha = 32.;
  block_.num += 1;
  ...
}
```

Note that the external name established by the C routine must end in underscore to link with the block created by the Fortran program.

Sharing I/O Between Fortran and C

Mixing Fortran I/O with C I/O (issuing I/O calls from both C and Fortran routines) is not recommended. It is better to do *all* Fortran I/O or *all* C I/O, but not both.

The Fortran I/O library is implemented largely on top of the C standard I/O library. Every open unit in a Fortran program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

If a Fortran main program calls C to do I/O, the Fortran I/O library must be initialized at program startup to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C function must take the Fortran I/O environment into consideration to perform I/O on open file descriptors.

However, if a C main program calls a Fortran subprogram to do I/O, the automatic initialization of the Fortran I/O library to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout` is lacking. This connection is normally made by

a Fortran main program. If a Fortran function attempts to reference the `stderr` stream (unit 0) without the normal Fortran main program I/O initialization, output will be written to `fort.0` instead of to the `stderr` stream.

The C main program can initialize Fortran I/O and establish the preconnection of units 0, 5, and 6 by calling the `f_init()` Fortran 77 library routine at the start of the program and, optionally, `f_exit()` at termination.

Remember: even though the main program is in C, you should link with `f77`.

Alternate Returns

Fortran's alternate returns mechanism is obsolescent and should not be used if portability is an issue. There is no equivalent in C to alternate returns, so the only concern would be for a C routine calling a Fortran routine with alternate returns.

The Sun Fortran implementation returns the `int` value of the expression on the `RETURN` statement. This is superbly implementation dependent and its use is not recommended:

Code Example 11-14 Alternate Returns (Obsolete)

C calls Fortran	Running the Example

`int altret_ ( int * );`
`main ()`
`{`
`  int k, m ;`
`  k =0;`
`  m = altret_( &k ) ;`
`  printf( "%d %d\n", k, m);`
`}`
`----------`
`SUBROUTINE ALTRET( I, *, *)`
`  INTEGER I`
`  I = I + 1`
`  IF(I .EQ. 0) RETURN 1`
`  IF(I .GT. 0) RETURN 2`
`  RETURN`
`END`
	`demo%` **`cc -c tst.c`**
	`demo%` **`f77 -o alt alt.f tst.o`**
	`alt.f:`
	`  altret:`
	`demo%` **`alt`**
	`1 2`
	
	The C routine receives the return value 2 from the Fortran routine because it executed the RETURN 2 statement.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z