sun microsystems

Sun FORTRAN Release Notes

Part No: 800-3139-10
Revision A of 27 January 1989
Release 1.2

Sun FORTRAN Release Notes

Contents

Tables

Preface

Purpose and Audience

This manual describes the changes and extensions in Sun FORTRAN 1.2. We assume you are familiar with the previous Sun FORTRAN and the SunOS™ file system. See also the *Sun FORTRAN Programmer's Guide.*

Conventions in Text

Note the following conventions we use in this manual to display information:

- `Plain typewriter font` indicates commands, prompts, and programming statements.
- **`Bold typewriter font`** indicates user input.
- *Italics* indicates general arguments or parameters that you should replace with the appropriate input. Italics are also used to indicate emphasis.
- Examples of coding are set in white boxes. Examples with user interaction are set in gray boxes to indicate the workstation screen. For example:

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

The basic SunOS prompt is merely the percent sign (%). However, most Sun workstations have distinct host names and our examples are more easily distinguished if we use a symbol longer than a % sign. For this reason, examples in this manual use `demo%` to denote the system prompt.

Organization

This manual consists of the following parts:

Chapter 1 is an overview of the changes and extensions.

Chapter 2 describes the changes and features.

Appendix A lists errata and addenda.

1

Introduction

1

Introduction

1.1. Overview

This release adds various features and improvements to Sun FORTRAN.

1.2. Revision Components

The new release consists of the previous Sun FORTRAN compiler and library with extensions added.

- The logical right shift function `LRSHIFT`
- Variable expressions in `FORMAT` statements
- The `OPTIONS` statement
- Initializing variables in type statements
- Initializing fields of structured records
- The intrinsic function `ZEXT`
- The VMS FORTRAN system routines

  `DATE`, `IDATE`, `SECNDS`, `TIME`, `RAN`, and `MVBITS`

1.3. Relationship with Other Sun Products

This Sun FORTRAN executes on all Sun-3™ and Sun-4™ systems under SunOS 4.0 and later.

2

New Features

2

New Features

2.1. The Logical Right Shift Pseudo-function LRSHFT

Sun FORTRAN now has a logical right shift pseudo-function, LRSHFT, that compiles into inline code.

The general form is:

```
LRSHFT( a1, a2 )
```

where LRSHFT shifts *a1* logically right by *a2* bits.

The shift functions are summarized here:

LSHIFT shifts a1 logically left by a2 bits. (inline code)
LRSHFT shifts a1 logically right by a2 bits. (inline code)
RSHIFT shifts a1 arithmetically* right by a2 bits. (inline code)
ISHIFT shifts a1 logically right if a2 > 0 and left if a2 < 0.

2.2. Variable Expressions in FORMAT Statements

In general, inside a FORMAT statement, any integer constant can be replaced by an arbitrary expression; the single exception is the "*n*" in an "*n*H..." edit descriptor. The expression itself must be inclosed in angle brackets.

For example, the "6" in:

```
1   FORMAT( 3F6.1 )
```

can be replaced by the variable "N", as in:

```
1   FORMAT( 3F<N>.1 )
```

or by the slightly more complicated expression "2*N+M", as in:

```
1   FORMAT( 3F<2*N+M>.1 )
```

Similarly, the "3" or "1" can be replaced by any expression.

---

* The *Sun FORTRAN Programmer's Guide*, page 238, describes RSHIFT as doing a *logical* right shift. That is an error. RSHIFT does an *arithmetic* right shift.

Rules and Restrictions for Variable Expressions in `FORMAT` 's

- The expression is reevaluated each time it is encountered during a format scan.
- If necessary, the expression is converted to integer type.
- Any valid FORTRAN expression is allowed, including function calls.
- Variable expressions are not allowed in formats generated at runtime.

2.3. The `OPTIONS` Statement

The `OPTIONS` statement overrides certain compiler command-line options. The general form is:

```
OPTIONS /qualifier [/qualifier ...]
```

The `OPTIONS` statement qualifiers recognized by Sun FORTRAN are:

Qualifier	*Action Taken*
`/[NO]G_FLOATING`	None (not implemented)
`/[NO]I4`	Enables/Disables the **`-i2`** option
`/[NO]f77`	Enables/Disables the **`-66`** option
`/CHECK=ALL`	Enables the **`-C`** option
`/CHECK=[NO]OVERFLOW`	None (not implemented)
`/CHECK=[NO]BOUNDS`	Enables the **`-C`** option
`/CHECK=[NO]UNDERFLOW`	None (not implemented)
`/CHECK=NONE`	Disables the **`-C`** option
`/NOCHECK`	Disables the **`-C`** option
`/[NO]EXTEND_SOURCE`	Disables the **`-e`** option

Rules and Restrictions for `OPTIONS` Statements

- The `OPTIONS` statement must be the *first* statement in a program unit. Note that this means it must be before the `BLOCK DATA`, `FUNCTION`, `PROGRAM`, and `SUBROUTINE` statements.
- The options set by the `OPTIONS` statement override the values from the compiler command-line.
- The options set by the `OPTIONS` statement endure for that program unit only.
- A qualifier can be abbreviated to four or more characters.
- Upper or lower case is not significant.

2.4. Initializing in Type Declarations

You can initialize variables in a typed data declaration, as in a DATA statement. The general form is:

type VariableName / constant / ...

or

type ArrayName / constant, ... /

or

*type ArrayName / r*constant /*

where *r* is a *repeat factor*.

For example:

```
CHARACTER LABEL*12 / "Standard" /
COMPLEX   STRESSPT / ( 0.0, 1.0 ) /
INTEGER   COUNT / 99 /,   Z / 1 /
REAL      PRICE / 0.0 /,  COST / 0.0 /
REAL      LIST(8)  / 0.0, 6*1.0, 0.0 /
```

Rules and Restrictions for Data Type Initialization

- For a simple variable, there must be exactly one constant.
- If any element of an array is initialized, all must be.
- You can use an integer as a *repeat factor*, followed by an asterisk (*), followed by a constant. (In the example above, six values of 1.0 are stored into array elements 2, 3, 4, 5, 6, and 7 of LIST.)
- If a variable or array is declared AUTOMATIC, then it cannot be initialized.
- A pointer-based variable or array cannot be initialized. For example, with:

```
INTEGER a / 4 /
POINTER ( x, a )
```

  You get a compiler warning message, and a does *not* get initialized.

- If a variable or array is not initialized, its value(s) are undefined.

2.5. Initializing Fields of Record Structures

You can initialize fields in a structured record, as long as the field declaration is a typed data declaration. For example:

```
STRUCTURE /PRODUCT/
    INTEGER*4     ID / 99 /
    CHARACTER*16 NAME
    CHARACTER*8   MODEL
    REAL*4        COST
    REAL*4        PRICE
END STRUCTURE
```

Every record that is declared to have the structure `PRODUCT` will have its `ID` field initialized to `99`. For example:

```
RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
```

This puts `99` into the `ID` field of records `CURRENT`, `PRIOR`, `NEXT`, and each of the 10 records of the array `LINE`.

Rules and Restrictions for Field Initialization

- The structure can be a substructure or union.
- If a record is declared `AUTOMATIC`, then its fields cannot be initialized.
- A pointer-based record or record field cannot be initialized. For example, with:

```
STRUCTURE / prod /
    INTEGER*4 i / 4 /
    CHARACTER*1 tag
END prod
RECORD / prod / a, b
POINTER ( x, a )
```

  You get a compiler warning message, and `a.i` does *not* get initialized.

- If a field is unnamed or not initialized, its value is undefined.

2.6. The Zero Extend Function ZEXT

The following zero-extend functions are now recognized by Sun FORTRAN. The first unused high-order bit is set to zero and extended toward the higher-order end to the width indicated in the table.

Table 2-1 *Zero-Extend Functions*

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
ZEXT	generic	zero-extend	–	–
IZEXT	specific	zero-extend	LOGICAL*1 or INTEGER*2	INTEGER*2
JZEXT	specific	zero-extend	LOGICAL or INTEGER	INTEGER*4

2.7. The VMS FORTRAN System Routines

These routines provide compatibility with VMS FORTRAN system routines.

Usage

To use these routines you must include the **-lV77** option on the **f77** command line, in which case you will also get the VMS versions of IDATE and TIME, instead of the Sun versions. For example:

```
demo% f77 myprog.f -lV77
```

Summary

Table 2-2 *Summary: VMS FORTRAN System Routines*

Name	*Definition*	*Calling Sequence*	*Argument Type*	*Returned Type*
DATE	Date as dd-mmm-yy	CALL DATE(c)	CHARACTER*9	n/a
IDATE	Date as d, m, y	CALL IDATE(d, m, y)	INTEGER	n/a
SECNDS	Time of day or elapsed time	t = SECNDS(u)	REAL	REAL
TIME	Current time as hhmmss	CALL TIME(t)	CHARACTER*8	n/a
RAN	Random number	r = RAN(s)	INTEGER*4	REAL
MVBITS	Move bit field	CALL MVBITS(src,i1,n, des,i2)	INTEGER	n/a

The error condition subroutine ERRSNS is *not* provided on Sun systems because it is totally specific to the VMS operating system. The terminate program subroutine EXIT was already provided by SunOS.

DATE

Get the current system date as a *character* string.

Usage

```
CALL DATE( c )
```

where:

c is a variable, array, array element, or character substring of type CHARACTER*9.

The form of the returned string c is dd-mmm-yy, where:

dd is the *day of the month*, as a 2-digit integer.

mmm is the *month* name as a 3-letter abbreviation.

yy is the *year*, as a 2-digit integer.

Example

```
demo% cat dat1.f
* dat1.f -- Get the date as a character string.
        CHARACTER c*9
        CALL DATE ( c )
        WRITE(*,"(' The date today is: ', A9 )" ) c
        END
demo% f77 dat1.f -lV77
dat1.f:
 MAIN:
demo% a.out
 The date today is: 23-Sep-88
demo%
```

IDATE

Get the current system date as three integers for month, day, and year.

Usage

```
CALL IDATE( m, d, y )
```

where m, d, and y are variables of type INTEGER.

and where:

m is the *month.*

d is the *day.*

y is the *year.*

Example

```
demo% cat ida1.f
* ida1.f -- Get the date as three integers m, d, y.
        INTEGER m, d, y
        CALL IDATE ( m, d, y )
        WRITE (*, "(' The date is: ',3i3)" ) m, d, y
        END
demo% f77 ida1.f -lV77
ida1.f:
 MAIN:
demo% a.out
 The date is:   9 23 88
demo%
```

SECNDS

Get the system time in seconds, minus the value of the argument.

Usage

```
t = SECNDS( t0 )
```

where:

`t0` is a constant, variable, or array element of type `REAL`, and `SECNDS` returns a value of type `REAL`. The returned value is the number of seconds since midnight, minus the argument supplied by the user.

Example

```
demo% cat sec1.f
      REAL  elapsed, t0, t1, x, y
      t0 = 0.0
      t1 = SECNDS( t0 )
      y  = 0.1
      DO I = 1, 1000
          x = ASIN( y )
      END DO
      elapsed = SECNDS( t1 )
      WRITE ( *, 1 )  elapsed
 1    FORMAT ( ' 1000 arcsines: ', F12.6, ' sec' )
      END
demo% f77 sec1.f -lV77
sec1.f:
 MAIN:
demo% a.out
 1000 arcsines:      6.699141 sec
demo%
```

Remarks

- The returned value from `SECNDS` is accurate to 0.01 second.
- The value is the number of seconds from midnight, and it correctly spans midnight.
- Some precision may be lost for small time intervals near the end of the day.

TIME

Get the current system time as a CHARACTER string.

Usage

```
CALL TIME( t )
```

where t, is a variable, array, array element, or character substring, and is of type CHARACTER*8.

The string returned is of the form hh:mm:ss, where each of hh, mm, and ss are 2-digits, and where:

hh is the *hour*.

mm is the *minute*.

ss is the *second*.

Example

```
demo% cat tim1.f
* tim1.f -- Get current time as a character string.
        CHARACTER  t*8
        CALL TIME( t )
        WRITE ( *, "(' The time is: ', A8 )" )  t
        END
demo% f77 tim1.f -lV77
tim1.f:
 MAIN:
demo% a.out
 The time is: 08:14:13
demo%
```

RAN

Generate a random number between 0 and 1; repeated calls to RAN generate a sequence of random numbers with a uniform distribution.

Usage

```
r = RAN( i )
```

where: `r` is a variable of type `REAL`, and `i` is a variable or array element of type `INTEGER*4`.

Example

```
demo% cat ran1.f
* ran1.f -- Generate random numbers.
        INTEGER i, n
        REAL  r(10)
        i = 760013
        DO n = 1, 10
                r(n) = RAN ( i )
        END DO
        WRITE ( *, "( 5 F11.6 )" )  r
        END
demo% f77 ran1.f -lV77
ran1.f:
 MAIN:
demo% a.out
   0.222058   0.299851   0.390777   0.607055   0.653188
   0.060174   0.149466   0.444353   0.002982   0.976519
demo%
```

Remarks

- The range includes 0.0 and excludes 1.0 .
- The algorithm is a multiplicative congruential type general random number generator.
- In general, the value of `i` is set *once* during execution of the calling program.
- The initial value of `i` should be a large odd integer.
- Each call to `RAN` gets the next random number in the sequence.
- To get a different sequence of random numbers each time you run the program, you must set the argument to a different initial value for each run.
- The argument is used by `RAN` to store a value for the calculation of the next random number according to the following algorithm:

```
SEED = 6909 * SEED + 1 (MOD 2**32)
```

- `SEED` contains a 32-bit number, and the high-order 24 bits are converted to floating point, and that value is returned.

MVBITS

Move a bit field

Usage

```
CALL MVBITS( src, ini1, nbits, des, ini2 )
```

where `src`, `ini1`, `nbits`, `des`, and `ini2`, are of type `INTEGER`, and where:

- `src` is the variable or array element that is the *source*.
- `ini1` is an expression for the *initial bit position* in the source.
- `nbits` is the *number of bits* to move.
- `des` is the variable or array element that is the *destination*.
- `ini2` is an expression for the *initial bit position* in the destination.

Example

```
demo% cat mvb1.f
* mvb1.f -- From src, initial bit 0, move 3 bits
*           to   des, initial bit 3.
*         src     des
*      543210  543210  <-- Bit numbers (VMS convention)
*      000111  000001  <-- Values before move
*      000111  111001  <-- Values after move
       INTEGER src, ini1, nbits, des, ini2
       DATA    src, ini1, nbits, des, ini2
&             /   7,    0,     3,   1,    3 /
       CALL MVBITS ( src, ini1, nbits, des, ini2 )
       WRITE (*,"(5o3)") src, ini1, nbits, des, ini2
       END
demo% f77 mvb1.f -lV77
mvb1.f:
 MAIN:
demo% a.out
  7  0  3 71  3
demo%
```

Remarks

- Bits are numbered according to VMS conventions: from low-ordered end (see example above).
- `MVBITS` changes only bits `ini2` through `ini2+nbits-1` of the `des` location, and no bits of the `src` location.
- Restrictions: *ini* 1+*nbits* <32 and *ini* 2+*nbits* ≤32

A

Errata and Addenda

A

Errata and Addenda

The following pages are Errata and Addenda for the *Sun FORTRAN Programmer's Guide*, Part Number: 800-2163-10.

A.1. Using the Compiler

Pages 15-22

Action: Remove pages 15 through 22 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 15 through 22 that follow.

Description of Changes

In Section 3.4, added information related to the `FORTRANCASE` environment variable (old pages 15 and 16).

In Section 3.7, inserted the **`-dalign`** option (old page 18), for the **`-misalign`** option, moved the restriction to Sun-4 (old page 20), inserted the **`-pic/PIC`** options (old page 21), and inserted the optional optimizer file `optim` under **`-Qoption`**, and **`-Qproduce`** (old page 22).

Minor formatting corrections on pages 18-22.

Continuation Lines

The default maximum number of continuation lines is 19 (1 initial and 19 continuation). See the **Nl***n* option, below.

Extended Lines

The compiler includes an option to accept extended source lines, with up to 132 characters. By default, it ignores any characters after column 72. To specify the recognition of extended source lines, use the **-e** option, as in this example:

```
demo% f77 -e prog.f
```

Padding

Padding is significant in lines such as:

```
        1         2         3         4         5         6         7
C23456789012345678901234567890123456789012345678901234567890123456789012

      DATA SIXTYH/60H
     1         /
```

3.4. Upper and Lower Case

In the standard, there are only 26 letters — FORTRAN 77 is a one-case language. Consistent with ordinary UNIX system usage, this compiler accepts upper-case or lower-case input. That is, the program source file can be in either lower-case or upper-case, or any mixture, but note the following general rules concerning case:

- The normal action of the compiler is to maintain names of procedures and names of variables in lower-case. (The **-U** option prevents this. In this **-U** mode, it is possible to specify external names with upper-case letters in them, and to have distinct variables differing only in case.)
- The compiler does not translate characters inside character-string constants.
- By default, the strings returned by `INQUIRE` are in upper-case.
- The debugger `dbxtool` does not convert to lower-case.

Case with `INQUIRE`

Since strings returned by `INQUIRE` are, by default, in upper-case, use of `INQUIRE` needs some caution regarding case. For example:

```
demo% cat inq1.f
* inq1.f Inquire with UPPER and lower-case
        CHARACTER ANSWER*15
        INQUIRE ( 6, SEQUENTIAL=ANSWER )
        IF ( ANSWER .EQ. 'YES' ) PRINT *, 'CAPS MATCH'
        IF ( ANSWER .EQ. 'yes' ) PRINT *, 'lowers match'
        END
demo% f77 inq1.f
inq1.f:
   MAIN:
demo% a.out
  CAPS MATCH
demo%
```

The match on *upper*-case is successful; the match on *lower*-case fails. You should probably be alert to such distinctions.

Case of Run-time Strings

You can use the environment variable `FORTRANCASE` to set upper or lower case for strings generated at run-time (logical variables, the `E` in floating point numbers, and strings returned by the `INQUIRE` statement). For example:

```
demo% setenv FORTRANCASE lower
```

Case with `dbxtool`

Use of the debugger `dbxtool` also requires some caution regarding case.* If your source file is in upper-case, then before you use `dbxtool` you should either tell `f77` to use upper-case, for example:

```
demo% f77 -U inq1.f
inq1.f:
   MAIN:
demo%
```

or use the `tr` command to translate the source file from upper-case to lower-case or vice versa. For example, to read the upper-case source file `SBENCH.f` and write the lower-case source file `sbench.f`:

```
demo% tr A-Z a-z < SBENCH.f > sbench.f
```

If your programs have bugs, `dbxtool` is useful. Most who have tried it found it was more than worth the bother of recompiling with the **`-U`** option.

3.5. Routines per File

The scope of FORTRAN variables and routines (as compared with C) has nothing to do with the files they reside in, so a source file can contain any number of compilation units (main programs, functions, or subroutines). However, there are two good reasons to keep each compilation unit in a separate source file:

- Reduce the compilation overhead of changing one procedure.
- Minimize loading of unreferenced functions.

Note that this applies only to the `.o` modules in libraries. Files explicitly named in the link command are unconditionally loaded.

`f77` produces one `.o` file for each `.f` file it processes. If any routine in the `.o` file is referenced, the linker `ld` copies in the entire `.o` file, loading all routines, referenced or not.

For example, suppose we have two files: `subs.f` and `main.f`:

File `subs.f` has routines `a` and `b`.
File `main.f` has a main program that calls `a` but not `b`.

* This debugger displays variable names so the users can *select* the variable they want displayed. It gets the variable names from the source file, so if the source has them in *upper* and the compiler has them in *lower*, then `dbxtool` cannot find the selected variable.

The command:

```
demo% f77 main.f sub.f
```

produces an `a.out` file that contains the code for subroutine `b` even though `b` is not referenced.

The `fsplit` command can be used to break up multiple-routine source files into a series of files, one routine per file.

3.6. Other Files

The `f77` command recognizes several other kinds of files. The table below summarizes the filename extensions that `f77` understands.

Table 3-1 *Filename Suffixes Sun FORTRAN Understands*

Suffix	*Language*	*Action*
.f	FORTRAN	Compile FORTRAN source files, put object files in current directory, default name of object file is that of the source but with `.o` suffix.
.F	FORTRAN	Process FORTRAN source files by the C preprocessor before compiling by `f77`.
.c	C	C source files are compiled by the C compiler. The `f77` and `cc` commands generate slightly different loading sequences, since FORTRAN programs need a few extra libraries and a different startup routine than do C programs.
.s	Assembler	Process assembly-language source files by the assembler `as`.
.il	In-line Expansion	Process in-line expansion code template files. These are used to expand calls to selected routines in-line when the **-O** option is used.
.o	Object Files	Pass object files through to the linker.

Note: Files with none of the above filename suffixes are passed to the linker.

Language Preprocessor

The `cpp` program is the C language preprocessor, which is invoked during the first pass of a FORTRAN compilation if the source filename has the `.F` extension. Its main uses here are for constant definitions and conditional compilation. See `cpp` (1), or the **-D***name* option in *Compiler Options*, in the next section.)

3.7. Compiler Options

The list below contains the options that `f77` understands. Note that the compiler option **-help** displays essentially the same list, as does the **man f77** command. (See the *Manual Pages*, online or in the appendices.)

-66 Report non-FORTRAN 66 constructs as errors.

-a Insert code to count how many times each basic block is executed. Invokes a runtime recording mechanism that creates a `.d` file for every `.f` file (at normal termination). The `.d` file accumulates execution data for the corresponding source file. The `tcov`(1) utility can then be run on the source file to generate statistics about the program.

-align *_block_*
Cause the common block whose FORTRAN name is *block* to be page-aligned: its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. For example, the command "**f77 -align _BUFFO_ GROWTH.F**" causes `BUFFO` to be page-aligned. This option applies to *uninitialized* data only: if any variable of the common block is initialized in a `DATA` statement, then the block will not be aligned. This option is passed to the linker.

-ansi
Identify all non-ANSI extensions. Note that `f77cvt` provides an option to flag any Sun FORTRAN extensions that it uses during the conversion of a VMS FORTRAN source file.
For more on `f77cvt`, see Section 10.4 — "The Source Code Converter."

-c Suppress linking and produce a `.o` file for each source file.

-C Compile code to check that subscripts are within declared array bounds.

-dalign
Sun-4™ only. Generate double load/store instructions wherever possible for faster execution. Using this option automatically triggers the **-f** option (see below) to cause all double typed data to be double aligned. With **-dalign**, you may not get ANSI standard FORTRAN alignment — a tradeoff of portability for speed. See also "Shared Libraries" in *Programming Utilities and Libraries*.

-dryrun
Show but do not execute commands constructed by the compiler driver.

-D*name=def*
-D*name*
Define *name* to the C preprocessor, as if by "`#define`". If no definition is given, the name is defined as "1" (`.F` files only).

-e Accept extended source lines, up to 132 characters long.

-f Align local data and `COMMON` blocks on 8-byte boundaries. Resulting code may not be standard and may not be portable.

-f*float_option*

Sun-2™ or Sun-3™ only. See the *Sun Floating-Point Programmer's Guide.*

-f68881
Generate code that assumes the presence of the Sun Floating-Point Accelerator (Sun-3 only).

-ffpa
Generate code that assumes the presence of the Sun-3 floating-point accelerator board (Sun-3 only).

-fsky
Generate code that assumes the presence of a Sky™ Floating-Point Processor board. Programs compiled with this option can only be run in systems that have a Sky board installed. (Sun-2 only).

-fsoft
Generate code that uses software floating-point calls (this is the default).

-fstore
Insure that expressions allocated to extended-precision registers are rounded to storage precision whenever an assignment occurs in the source code. Only has effect when **-f68881** is specified (Sun-3 only).

-fswitch
Run-time-switched floating-point calls. The compiled object code is linked at runtime to routines that support the FPA, MC68881, Sky floating-point board, or software-floating-point calls, depending on the system that is running the program (Sun-2 or Sun-3).

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

-g Produce additional symbol table information for `dbx` or `dbxtool`. Also, pass the `-lg` file to `ld` (1).

-help Display an equivalent of this list of options.

-i2 Make the default size of integer and logical constants and variables short (2 bytes).

-i4 Make the default size of integer and logical constants and variables four bytes (this is the default).

-I*path*

Add *path* to the list of directories in which to search for '`#include`' files with *relative* pathnames (not beginning with */*). Search first for '`#include`' files whose names do not begin with '/' in the directory containing the source file, then in directories named in **-I** options, and finally in directories on a standard list (`.F` suffix files only). Note that this does not affect FORTRAN's `INCLUDE` statement, only the C preprocessor's. For example, "**f77 -I/usr/applib growth.f**" causes the compiler to search for '`#include`' files in `/usr/applib`.

-l*x* Link with object library `/lib/lib`*x*`.a`, where *x* is a string. If that does not exist, then `ld` tries `/usr/lib/lib`*x*`.a` (see `ld` (1)).

-L*dir*
Add *dir* to the list of directories containing object-library routines (for linking using `ld`(1)).

-misalign
Sun-4 only. Allow for misaligned data in memory. Use this option *only* if you get a warning that `COMMON` or `EQUIVALENCE` statements cause data to be misaligned. **WARNING:** With this option, the compiler will generate much *slower* code for references to dummy arguments. If you can, you should recode the indicated section instead of recompiling with this option. For example, the program

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

causes the error message

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

-N[**cdlnqsx**]*nnn*
Make static tables in the compiler bigger. `f77` complains if tables overflow and suggests you apply one or more of these flags. These flags have the following meanings:

- **c** Maximum depth of nesting for control statements (for example, `DO` loops). The default is 20.
- **d** Maximum depth of nesting for data structures and unions. The default is 20.
- **l** Maximum number of continuation lines for a continued statement. The default is 19 (1 initial and 19 continuation).
- **n** Maximum number of identifiers. The default is 1009.
- **q** Maximum number of equivalenced variables. The default is 150.
- **s** Maximum number of statement numbers. The default is 401.
- **x** Maximum number of external names (common block names, subroutine and function names). The default is 200.

-o *output*
Name the final output file *output* instead of `a.out`.

-onetrip
Compile DO loops so that they are performed at least once if reached. Sun FORTRAN DO loops are not performed at all if the upper limit is smaller than the lower limit, unlike FORTRAN 66 DO loops.

-O*n* Optimize the object code. If you use **-g**, then **-O***n* is ignored.

-O1 Peephole optimization only. Do not use **-O1** unless **-O2** and **-O3** result in excessive compilation time, or running out of swap space.

-O2 Partial optimization. Does a restricted set of global optimizations. Do not use **-O2** unless **-O3** results in excessive compilation time, or running out of swap space.

-O3 Global Optimization. (same as **-O**)

Note:
If the optimizer runs out of swap space, try any of the following possibly corrective measures (listed in increasing order of difficulty):

- Change from **-O3** to **-O2**.
- Divide large, complicated routines into smaller, simpler ones.
- Increase the limit for the stacksize: insert the line "`limit stacksize 8 megabytes`" into your `.cshrc` file.
- Repartition your disk with two to four times as much swap space. Backup everything first. You may well need help from your system administrator to do this.

-p Prepare object files for profiling, see `prof` (1).

-pg Produce counting code in the manner of **-p**, but invoke a runtime recording mechanism that keeps more extensive statistics and produces a `gmon.out` file at normal termination. An execution profile can then be generated by use of `gprof` (1).

-pic
Produce position-independent code. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 64K on MC68000-family processors, or to 8K on SPARC processors.

-PIC
Similar to **-pic**, but allows the global offset table to span the range of 32-bit addresses. This is for use in those rare cases where there are too many global data objects for **-pic**.

-pipe
Use pipes, rather than intermediate files between compilation stages. Very cpu-intensive.

-P Partial optimization. (same as **-O2**)

-Qoption *prog opt*
Pass the option *opt* to the program *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of: `as`, `c2`, `cg`, `cpp`, `f77pass1`, `iropt`, `inline`, `ld`, or `optim`.*

-Qpath *pathname*
Insert the directory *pathname* into the compilation search path (to use alternate versions of programs invoked during compilation). This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver (such files as `*crt*.o` and `bb_link.o`).

-Qproduce *sourcetype*
Produce source code of the type *sourcetype*, where *sourcetype* is one of:
`.o` Object file from `as` (1).
`.s` Assembler source (from `f77pass1`, `inline`, `c2`, `cg`, or `optim`.)*

-S Compile the named programs, and leave the assembly-language output on corresponding files suffixed with `.s` (no `.o` file is created).

-temp=*dir*
Set directory for temporary files to be *dir*.

-time
Report execution times for the various compilation passes.

-u Make the default type of variables 'undefined', rather than using FORTRAN implicit typing.

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

-v Print the name of each pass as the compiler executes.

-w Suppress all warning messages.

-w66
Suppress only messages generated by programs using obsolete FORTRAN 66 features.

Unrecognized Arguments

Other arguments are taken to be either linker option arguments or names of `f77`-compatible object programs, typically produced by an earlier run, or perhaps libraries of `f77`-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program called (by default) `a.out` or with a filename specified by the **-o** option.

* For a Sun-2, Sun-3, or Sun-4, the optimizer file is `iropt`; for a Sun386*i*,™ it is `optim`.

A.2. Data Structures and Expressions

Pages 43 and 44

Action: Remove pages 43 and 44 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 43 and 44 that follow.

Description of Changes

Formatting corrections on page 43.

- The only statements allowed between the STRUCTURE statement and the END STRUCTURE statement are *field-declaration* statements and PARAMETER statements. A PARAMETER statement inside a structure declaration block is equivalent to one outside.

Rules and restrictions for fields

Fields that are type declarations use the identical syntax of normal FORTRAN type statements, and all Sun FORTRAN types are allowed, subject to the following rules and restrictions:

- Any dimensioning needed must be in the type statement. The DIMENSION statement has no effect on field names.
- You can specify the pseudo-name %FILL for a field name to align fields in a record.
- You must explicitly type all field names. The IMPLICIT statement does not apply to statements in a STRUCTURE declaration, nor do the implicit I,J,K,L,M,N rules apply.
- You can't use arrays with adjustable or assumed size in field declarations, nor can you include passed-length CHARACTER declarations.
- **Field offsets** — In a structure declaration, the offset of field *n* is the offset of the preceding field, plus the length of the preceding field, possibly corrected for any adjustments made to maintain alignment. For a summary of storage allocation, see the Subsection "Storage Allocation" in Section 4.3 — "Data Types."

Record declaration

The RECORD statement declares variables to be records with a specified structure, or declares arrays to be arrays of such records. The syntax of a RECORD statement is as follows:

```
RECORD /structure-name/ record-list
     [,/structure-name/ record-list]
          .
          .
     [,/structure-name/ record-list]
```

where *structure-name* is the name of a previously declared structure, and *record-list* is a list of variables, arrays, or arrays with dimensioning and index ranges, separated by commas.

For example, using the structure in the example above:

```
RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
```

Each of the three variables CURRENT, PRIOR, and NEXT is a record which has the PRODUCT structure, and LINE is an array of 10 such records.

Rules and restrictions for records

- Each record is allocated separately in memory.
- Initially, records have undefined values.
- Records, record fields, record arrays, and record-array elements are allowed as arguments and dummy arguments. When you pass records as arguments, their fields must match in type, order, and dimension. The record declarations in the calling and called procedures must match. Within a union declaration, the order of the map fields is not relevant — see "Unions and maps," later in this section.
- Records and record fields are allowed in `COMMON` and `DIMENSION` statements.
- Records and record fields are not allowed in `DATA`, `EQUIVALENCE`, `NAMELIST`, or `SAVE` statements.

Record and field reference

You can refer to a whole record, or to an individual field in a record, and since structures can be nested, a field can itself be a structure, so you can refer to fields within fields, within fields, etc. The syntax of record and field reference is as follows:

record-name [.*field-name*] ... [.*field-name*]

where *record-name* is the name of a previously defined record variable, and each *field-name* is the name of a field in the record immediately to the left.

Examples of references are given below, based on the structure and records of the above two examples:

```
RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
...
CURRENT = NEXT
LINE(1) = CURRENT
WRITE ( 9 )  CURRENT
NEXT.ID = 82
```

In this example, the first assignment statement copies one whole record (all five fields) to another record, the second assignment statement copies a whole record into the first element of an array of records, the `WRITE` statement writes a whole record, and the last statement sets the `ID` of one record to 82.

A complete sample program is listed below to show structure and record declarations, record and field assignments, and field output:

A.3. Input and Output

OPEN

Action: Remove pages 79 and 80 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 79 and 80 that follow.

Description of Changes

On page 79, clarified opening sequential access files with `RECL`.

ACCESS Optional character expression; one of: APPEND, DIRECT, or SEQUENTIAL. The default is SEQUENTIAL.

If ACCESS='APPEND' is specified:

- SEQUENTIAL and FILEOPT='EOF' are assumed. This is for opening a file to append records to an existing sequential-access file. This is a Sun FORTRAN extension.

If ACCESS='DIRECT' is specified:

- RECL must also be given, since all I/O transfers are done in multiples of fixed-size records.
- Only directly accessible files are allowed; thus, tty, pipes, and magnetic tape are not allowed.
- If FORM is not specified, unformatted transfer is assumed.
- If FORM='UNFORMATTED', the size of each transfer depends upon the data transferred.

If ACCESS='SEQUENTIAL':

- RECL is ignored; a runtime warning is issued. The ANSI standard prohibits RECL for sequential access.
- No padding of records is done.
- Files don't have to be randomly accessible; thus tty, pipes, and tapes can be used.
- If FORM is not specified, formatted transfer is assumed.
- If FORM='FORMATTED', each record is terminated with a newline (\n) character. This means that each record actually has one extra character.
- If FORM='PRINT', the file acts like a FORM='FORMATTED' file, except for the interpretation of column-1 characters on output (0 = double space, 1 = form feed, and blank = single space).
- If FORM='UNFORMATTED', each record is preceded and terminated with an INTEGER*4 count, making each record 8 characters longer than normal. This convention is not shared with other SunOS programs, so is useful only for communicating between FORTRAN programs.

FORM An optional character expression. The options are 'FORMATTED', 'UNFORMATTED', or 'PRINT'.
If not specified, 'FORMATTED' is assumed.
Interacts with ACCESS.

RECL "RECL=*n*" specifies a record length of *n* characters.
Required if ACCESS='DIRECT'; ignored otherwise. See ACCESS='SEQUENTIAL' above.

Each WRITE defines one record and each READ reads one record (unread characters are flushed).

ERR — An optional clause, with an integer statement label to branch to if an error occurs during the OPEN.

IOSTAT — An optional clause, with an integer variable that receives the error status from an OPEN.

Note: If you want to avoid aborting the program when an error occurs on an OPEN, then include an ERR=*label* or an IOSTAT=*name* .

BLANK — An optional character expression that indicates how blanks are treated. For formatted input only; the options are 'ZERO' (blanks treated as zeroes), and 'NULL' (blanks ignored during numeric conversion). If not specified, 'NULL' is assumed.

STATUS — An optional character expression. Possible values are:

- 'OLD' — the file already exists (nonexistence is an error). For example: STATUS='OLD'
- 'NEW' — the file doesn't exist (existence is an error) **Note:** 'FILE=*name*' is required.
- 'UNKNOWN' — existence is unknown (the default).
- 'SCRATCH' — In general, if you open a file with STATUS='SCRATCH', then the file will be removed when it is closed.
  Note: The standard prohibits opening a named file as scratch, that is if the OPEN statement has a FILE=*name* option, then it cannot have a STATUS='SCRATCH' option. Sun FORTRAN allows opening named files as scratch, but such files will be removed when closed or at program termination unless there is an explicit CLOSE statement with the option STATUS='KEEP'.

FILEOPT — An optional character expression. The options are:

- 'NOPAD' — don't extend records with blanks if you read past the end-of-record (formatted input only). That is, a *short* record causes an abort with an error message, rather than just filling with trailing blanks and continuing.
- 'BUFFER=*n*' — This suboption is for magnetic tape only. It sets the size of the I/O buffer to use. It is necessary only when writing, since the I/O system defaults to 64K-character buffers for tape, allowing reads to anything smaller than that. **WARNING:** It must be at least 8 characters greater than the largest record you write to avoid spanning tape blocks.
- 'EOF' — opens a file at end-of-file rather than at the beginning (useful for appending data to the file).

Accessing Files

Action: Remove pages 99 and 100 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 99 and 100 that follow.

Description of Changes

On page 99, moved ampersands to column one.

- character expressions, such as:

```
      FILE=PREFIX(:LNBLNK(PREFIX)) // '/' //
     &     NAME(:LNBLNK (NAME)),...
```

Some ways a program can get filenames are:

- By reading from a file or terminal keyboard, such as:

```
READ( 4, 401) FILNAM
```

- From the command line, such as:

```
CALL GETARG( ARGNUMBER, FILNAM)
```

- From the environment, such as:

```
CALL GETENV( STRING, FILNAM )
```

The example below shows one way to construct a filename:

```
    CHARACTER*1024 FUNCTION FULLNAME ( NAME )
    CHARACTER*(*)  NAME
    CHARACTER*1024 PREFIX
C
C In  path names starting with '~/':
C  replace the tilde with the home directory name;
C  prefix relative pathname by path to current directory;
C  leave absolute path names unchanged.
C
    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
    END
```

Accessing Unnamed Files

When a program opens a FORTRAN file without a name, the runtime system supplies a filename. There are several ways it can do this.

Opened as scratch

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

Already open

If a FORTRAN program has a file already open, an `OPEN` statement that specifies only the file's logical unit number and the parameters to change can be used to change some of the file's parameters (specifically, `BLANK` and `FORM`). The system determines that it should not really `OPEN` a new file, but just change the parameter values. Thus, this looks like a case where the runtime system would make up a name, but it does not.

Other

In all other cases, the runtime system `OPEN`s a file with a name of the form `fort.n`, where *n* is the logical unit number given in the `OPEN` statement.

Passing filenames to programs

The SunOS file system does not have any notion of temporary filename binding (or file equating) as some other systems do. Filename binding is the facility that is often used to associate a FORTRAN logical unit number with a physical file without changing the program. This mechanism evolved to communicate filenames more easily to the running program, because in FORTRAN 66 you could not open files by name. With SunOS or with UNIX there are several satisfactory ways to communicate filenames to a FORTRAN program including command-line arguments and environment-variable values. For example, see the file `ioinit.f` in `libI77`, which is discussed in Section 7.2, under Subsection "Logical Unit Preattachment." The program can then use those logical names to open the files. The next section describes two additional ways to change a program's input and output files without changing the program, called *redirection* and *piping*.

Preconnected units

When a FORTRAN program begins execution under SunOS, there are usually three units already open. These are called *preconnected units*. Their names are *standard input*, *standard output*, and *standard error*. In FORTRAN programs:

- standard input is logical unit 5
- standard output is logical unit 6
- standard error is logical unit 0

All three are connected to your workstation or window, unless file redirection or piping is done at the command level.

Other units

All other units are preconnected to files named `fort.`*n* where *n* is the corresponding unit number. These files need not exist, and are created only if the units are actually used, and if the first action to the unit is a `WRITE` or `PRINT`. That is, only if an `OPEN` statement does not override the preconnected name before any `WRITE` or `PRINT` is issued for that unit. For example, the program:

```
WRITE( 15 ) 2
END
```

writes a single unformatted record on the `fort.15` file.

A.4. Program Development

Action: Remove pages 131 and 132 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 131 and 132 that follow.

Description of Changes

On page 132, moved ampersands to column one, and in the last format changed double quote to single quote.

Table 8-1 *Time Functions Available to FORTRAN*

Name	*Function*	*Man Page*
TIME	Returns the number of seconds elapsed since 1 January, 1970	`time(3f)`
FDATE	Returns the current time and date as a character string	`fdate(3f)`
IDATE	Returns the current month, day, and year in an integer array	`idate(3f)`
ITIME	Returns the current hour, minute, and second in an integer array	`itime(3f)`
CTIME	Converts time returned by `time` function to character string	`ctime(3f)`
LTIME	Converts time returned by `time` function to local time	`ltime(3f)`
GMTIME	Converts time returned by `time` function to Greenwich time	`gmtime(3f)`
ETIME	Returns elapsed user and system time for program execution	`etime(3f)`
DTIME	Returns elapsed user and system time since last call to `dtime`	`dtime(3f)`

The following program is an example of how to implement FORTRAN time functions that might appear on other systems:

```
      SUBROUTINE STARTCLOCK
      COMMON / MYCLOCK /  MYTIME
      INTEGER MYTIME
      INTEGER TIME
      MYTIME = TIME()
      RETURN
      END

      FUNCTION WALLCLOCK
      INTEGER WALLCLOCK
      COMMON / MYCLOCK /  MYTIME
      INTEGER MYTIME
      INTEGER TIME
      INTEGER NEWTIME
      NEWTIME = TIME()
      WALLCLOCK = NEWTIME - MYTIME
      MYTIME = NEWTIME
      RETURN
      END


      PROGRAM TESTTIME
C     Play with some system timing functions
      INTEGER WALLCLOCK, ELAPSED
      CHARACTER*24 GREETING
      REAL DTIME
      REAL TIMEDIFF, TIMEARRAY(2)

C     Print a heading
      CALL FDATE( GREETING )
      WRITE( 6, 10 ) GREETING
10    FORMAT('1   Hi, it''s ' , A24 /)
C     See how long an 'ls' takes, in seconds
      CALL STARTCLOCK
      CALL SYSTEM( 'ls' )
      ELAPSED = WALLCLOCK()
      WRITE( 6, 20 ) ELAPSED
20    FORMAT(//,'Elapsed time ', I4, ' seconds'///)
C     Now test the CPU time for some trivial computing
      TIMEDIFF = DTIME( TIMEARRAY )
      Q = 0.01
      DO 30 I = 1, 1000
          Q = ATAN( Q )
30    CONTINUE
      TIMEDIFF = DTIME( TIMEARRAY )
      WRITE( 6, 40 ) TIMEDIFF
40    FORMAT(//,'Computing ATAN(Q) 1000 times',
&           /  'took ', F6.3,' seconds.'/)
      END
```

A.5. The C—FORTRAN Interface

Action: Remove pages 183 and 184 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 183 and 184 that follow.

Description of Changes

On page 184, under "Return a float", revised description and examples to cover all Sun architectures.

Arrays

A C array always starts at zero, but a FORTRAN array starts at 1, by default, so in the above example, FORTRAN `B(2)` is equivalent to C `b[1]`. FORTRAN arrays are stored in column-major order, C arrays in row-major order.

Calling C from FORTRAN

The following examples illustrate FORTRAN programs that call C functions.

Repeat a character

The called function has the task of building a character string by repeating a character `N` times, where the character and `N` are arguments.

`main.f`:

```
      CHARACTER STRING*100, REPEAT*50
      STRING = REPEAT ( '*', 10 )
      PRINT *, STRING
      END
```

File `repeat.c`:

If `repeat` were a FORTRAN function, the compiler would hide the details of managing character strings; however, since repeat is written in C, the housekeeping must be explicit:

The compiler appends a trailing underscore to all external names in FORTRAN programs, so you need to add an underscore to the name of the C function called.

The function's returned character string is passed by the two extra arguments `retval_ptr` and `retval_len`, a pointer to the start of the string and the string's length.

The character-string argument is passed with `char_ptr` and char_len , the first points to the string start and the second gives the string's length. The repeat factor is passed as `n_ptr`.

```
#include <stdio.h>

repeat_(retval_ptr, retval_len, char_ptr, n_ptr, char_len)
char *retval_ptr, *char_ptr;
int  retval_len, *n_ptr, char_len;
{
    int count, i;
    char *cp;

    count = *n_ptr;
    if (count > retval_len) {
        fprintf( stderr, "repeat count too large\n");
        count = retval_len;
    }

    cp = retval_ptr;
    for (i=0; i<count; i++) {
        *cp++ = *char_ptr;
    }

    for (i=count; i<retval_len; i++) {
        *cp++ = ' ';
    }
}
```

This program can be compiled with the **`f77`** command:

```
demo% f77 main.f repeat.c
```

Since every character argument in the list is associated with an additional argument giving the string's length, such FORTRAN strings need not terminate with a null character, as required by C.

Return a `float`

If `MAIN` declares `REPEAT` as an `INTEGER`, `LOGICAL`, `REAL`, or `DOUBLE PRECISION` function, then the two initial arguments would not be present, so the return value could be passed back to the FORTRAN program with a `return` statement. To return a `float`, and have it work on all Sun architectures*, use the macros from the `math.h` header file, as in the following example:

```
* testincr1.f
      REAL  incr, R, S
      R = 1.0
      S = incr( R )
      PRINT *, S
      STOP
      END
```

```
/* incr1.c: return a float */
#include <math.h>
FLOATFUNCTIONTYPE incr_( float_ptr )
float *float_ptr ;
{
    float  f ;
    f = *float_ptr ;
    f++ ;
    RETURNFLOAT ( f ) ;
}
```

To return a *pointer* to a `float` (also works on all Sun architectures):

```
* testincr2.f
      POINTER   ( P, S )
      REAL R, S
      R = 1.0
      P = incr( R )
      PRINT *, S
      STOP
      END
```

```
/* incr2.c: return a pointer a float */
static float f ;
float *incr_( float_ptr )
float *float_ptr ;
{
    f = *float_ptr ;
    f++ ;
    return &f ;
}
```

Either one of the above pairs prints `2.000000`. For example:

```
demo% f77 testincr1.f incr1.c
testincr1.f:
testincr1.f:
 MAIN:
incr1.c:
Linking:
demo% a.out
   2.000000
demo%
```

* If C returns a `float`, C promotes it to a `double`; different architectures handle this differently.

A.6. f77 Man Pages

Action: Remove pages 253 through 258 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 253 through 258 that follow.

Description of Changes

Inserted the optional optimizer file `optim` under **-Qoption**, **-Qproduce**, and **FILES**.

Inserted the options **-dalign**, **-pic**, and **-PIC**.

Inserted the note that **-misalign** is restricted to the Sun-4.

NAME

intro – introduction to FORTRAN Manual Pages

DESCRIPTION

This section includes the man pages for **f77, f77cvt, fpr, fsplit**, and **ratfor**.

NAME

f77 – Sun FORTRAN compiler

SYNOPSIS

f77 [**–66**] [**–a**] [**–align** *_block*] [**–ansi**] [**–c**] [**–C**] [**–dalign**] [**–dryrun**] [**–D***name*[=*def*]] [**–e**]
[*float_option*] [**–fstore**] [**–f**] [**–F**] [**–g**] [**–help**] [**–i2**] [**–i4**]
[**–I***pathname*] [**–l***lib*] [**–L***dir*] [**–misalign**] [**–N**[**cdlnqsx**]*nnn*]
[**–o** *outfile*] [**–onetrip**] [**–O**[**123**]] [**–p**] [**–pg**] [**–pic**] [**–PIC**] [**–pipe**]
[**–Qoption** *prog opt*] [**–Qpath** *pathname*] [**–Qproduce** *sourcetype*] [**–S**] [**–temp**=*dir*]
[**–time**] [**–u**] [**–U**] [**–v**] [**–w**[**66**]] *sourcefile* ...

DESCRIPTION

f77 is the Sun FORTRAN compiler, which translates programs written in the Sun FORTRAN programming language into executable load modules or into relocatable binary programs for subsequent linking with **ld**(1). Sun FORTRAN is a superset of FORTRAN 77, with many extensions, including those to provide compatibility with VMS FORTRAN (in conjunction with **f77cvt**(1)). In addition to the many flag arguments (options), **f77** accepts several types of files.

Files with names ending in **.f** are taken to be Sun FORTRAN source files; they are compiled, and each object program is put in the current directory in a file with the same name as the source, with **.o** substituted for **.f**.

Files with names ending in **.F** are also taken to be Sun FORTRAN source files, but they are preprocessed by the C preprocessor (equivalent to a **cc –E** command) before they are compiled by the **f77** compiler.

Files with names ending in **.c** or **.s** are taken to be C or assembly source files and are compiled or assembled, producing **.o** files.

Files with names ending in **.il** are taken to be in-line expansion code template files; these are used to expand calls to selected routines in-line when the **–O** option is in effect.

Files with names ending in **.vf** or **.for** are assumed by the **f77cvt**(1) source code converter (not by the **f77** compiler) to be valid VMS FORTRAN source files and are converted to source files acceptable to both Sun FORTRAN and VMS FORTRAN compilers, except for possible VMS FORTRAN features which it can't convert, which are reported by error messages.

OPTIONS

See **ld**(1) for link-time options.

–66 Report non-FORTRAN 66 constructs as errors.

–a Insert code to count how many times each basic block is executed. Invokes a runtime recording mechanism that creates a **.d** file for every **.f** file (at normal termination). The **.d** file accumulates execution data for the corresponding source file. The **tcov**(1) utility can then be run on the source file to generate statistics about the program.

–align *_block_* Cause the common block whose FORTRAN name is *block* to be page-aligned: its size is increased to a whole number of pages, and its first byte is placed at the beginning of a page. This option is passed on to the linker; it's a linker option. For example, the command "`f77 -align _BUFFO_ GROWTH.F`" causes BUFFO to be page-aligned.

–ansi Identify all non-ANSI extensions. Note that **f77cvt** provides an option to flag any Sun FORTRAN extensions that it uses during the conversion of a VMS FORTRAN source file.

–c Suppress linking with **ld**(1) and produce a **.o** file for each source file. A single object file can be named explicitly using the **–o** option.

–C Compile code to check that subscripts are within the declared array bounds.

–dalign Generate double load/store instructions wherever possible to give faster execution. Using this option automatically triggers the **–f** option (see below) to cause all double typed data to be double aligned. Note that with the **–dalign** option, you may not get the usual alignment guaranteed by ANSI standard FORTRAN so you may be sacrificing portability to gain speed. See also "Shared Libraries" in *Programming Utilities and Libraries* . (Sun-4 only).

–dryrun Show but do not execute the commands constructed by the compilation driver.

–D*name* [*=def*] Define a symbol *name* to the C preprocessor, **cpp**(1). Equivalent to a **#define** directive in the source. If no *def* is given, *name* is defined as '1' (**.F** suffix files only).

–e Accept extended source lines, up to 132 characters long.

–f Align local data and common blocks on 8-byte boundaries. Resulting code may not be standard and may not be portable.

float_option Floating-point code generation option. This option does not apply to the Sun-4, which generates SPARC floating-point instructions. For the Sun-2 and Sun-3, *float_option* can be one of:

–f68881
Generate in-line code for the Motorola MC68881 floating-point coprocessor (Sun-3 only).

–ffpa Generate in-line code for the Sun-3 Floating-Point Accelerator board (Sun-3 only).

–fsky Generate in-line code for the Sky Floating-Point Processor (Sun-2 only).

–fsoft Generate software floating-point calls (Sun-2 and Sun-3 systems, for which this is the default).

–fstore Insure that expressions allocated to extended precision registers are rounded to storage precision whenever an assignment occurs in the source code. Only has effect when **–f68881** is specified. (Sun-3 only)

–fswitch
Generate runtime-switched floating-point calls. The compiled object code is linked at runtime to routines that support one of the above types of floating-point code. This was the default in previous releases. Only for use with programs that are floating-point intensive and which must be portable to machines with various floating-point options (Sun-2 or Sun-3).

–F Apply the C preprocessor to **.F** files. Put the result in corresponding **.f** files, but do not compile them. No linking is done.

–g Produce additional symbol table information for **dbx**(1) and pass the **–lg** flag to **ld**(1).

–help Display an equivalent of this list of options.

–i2 Make the default size of integer and logical constants and variables two bytes.

–i4 Make the default size of integer and logical constants and variables four bytes (this is the default).

–I*pathname* Add *pathname* to the list of directories in which to search for **#include** files with relative filenames (not beginning with /). The preprocessor first searches for **#include** files in the directory containing *sourcefile*, then in directories named with **–I** options (if any), and finally in **/usr/include/f77** (applies to processing of **.F** suffix files only).

–l*lib* Link with object library *lib* (for **ld**(1)).

–L*dir* Add *dir* to the list of directories containing object-library routines (for linking using **ld**(1)).

–misalign Sun-4 only. Allow for misaligned data in memory. Use this option **only** if you get a warning that **COMMON** or **EQUIVALENCE** statements cause data to be misaligned. WARNING: With this option, the compiler will generate very much *slower* code for references to dummy arguments. If you can, you should recode the indicated section instead of recompiling with this option.

–N[cdlnqsx]*nnn* Make static tables in the compiler bigger. **f77** complains if tables overflow and suggests you apply one or more of these flags. These flags have the following meanings:

- **c** Maximum depth of nesting for control statements (for example, **DO** loops). Default is 20.
- **d** Maximum depth of nesting for data structures and unions. Default is 20.
- **l** Maximum number of continuation lines for a continued statement. The default is 19 (1 initial and 19 continuation).
- **n** Maximum number of identifiers. Default is 1009.
- **q** Maximum number of equivalenced variables. Default is 150.
- **s** Maximum number of statement numbers. Default is 401.
- **x** Maximum number of external names (common block, subroutine, and function names). Default is 200.

Multiple **–N** options increase sizes of multiple tables.

–o *outfile* Name the output file *outfile*. *outfile* must have the appropriate suffix for the type of file to be produced by the compilation (see FILES, below). *outfile* cannot be the same as *sourcefile* (the compiler will not overwrite the source file).

–onetrip Compile **DO** loops so that they are performed at least once if reached. Otherwise, Sun FORTRAN **DO** loops are not performed at all if the upper limit is smaller than the lower limit.

–O[123] Optimize the object code. This invokes both the global intermediate code optimizer and the object code optimizer.

- **–O1** Peephole Optimization only. Do not use **–O1** unless **–O2** and **–O3** result in excessive compilation time, or running out of swap space.
- **–O2** Partial optimization. Does a restricted set of global optimizations. Do not use **–O2** unless **–O3** results in excessive compilation time, or running out of swap space. (Same as **–P**)
- **–O3** Global Optimization. (same as **–O**)

If the optimizer runs out of swap space, try any of the following possibly corrective measures (listed in increasing order of difficulty):

1. Change from **-O3** to **-O2 .**
2. Divide large, complicated routines into smaller, simpler ones.
3. Increase the limit for the stacksize: insert the line "**limit stacksize 8 megabytes**" into your **.cshrc** file.
4. Repartition you disk with two to four times as much swap space. Backup everything first. You may well need help from your system administrator to do this.

–p Prepare the object code to collect data for profiling with **prof**(1). Invokes a runtime recording mechanism that produces a **mon.out** file (at normal termination).

–pg Prepare the object code to collect data for profiling with **gprof**(1). Invokes a runtime recording mechanism that produces a **gmon.out** file (at normal termination).

–pic Produce position-independent code. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 64K on MC68000-family processors, or to 8K on SPARC processors.

–PIC Similar to **-pic**, but allows the global offset table to span the range of 32-bit addresses. This is for use in those rare cases where there are too many global data objects for **–pic**.

–pipe Use pipes, rather than intermediate files between compilation stages. Very cpu-intensive.

–P See the **–O2** option.

–Qoption *prog opt*
Pass the option *opt* to the program *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of: **as**, **c2**, **cg**, **cpp**, **f77pass1**, **iropt**, **inline**, **ld**, or **optim**.

–Qpath *pathname*
Insert directory *pathname* into the compilation search path (to use alternate versions of programs invoked during compilation). This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver (such files as ***crt*.o** and **bb_link.o**).

–Qproduce *sourcetype*
Produce source code of the type *sourcetype*, where *sourcetype* can be one of:
.o Object file from **as**(1).
.s Assembler source (from **f77pass1, inline, c2**, **cg**, or **optim**).

–S Compile the named programs, and leave the assembly language output on corresponding files suffixed **.s** (no **.o** file is created).

–temp=*dir* Set directory for temporary files to be *dir*.

–time Report execution times for the various compilation passes.

–u Make the default type of a variable 'undefined', rather than using the FORTRAN default rules.

–U Do not convert upper case letters to lower case. The default is to convert upper case letters to lower case, except within character string constants.

–v Verbose. Print the name of each pass as the compiler executes.

–w[**66**] Suppress all warning messages. **–w66** suppresses only FORTRAN 66 compatibility warnings.

Other arguments are taken to be either linker option arguments, or **f77**-compatible object programs, typically produced by an earlier run, or libraries of **f77**-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program in the file specified by the **–o** option, or in a file named **a.out** if the **–o** option is not specified.

ENVIRONMENT

FLOAT_OPTION When no floating-point option is specified, the compiler uses the value of this environment variable (if set). Recognized values are: **f68881**, **ffpa**, **fsky**, **fswitch** and **fsoft**.

FILES

a.out	executable output file
file.**a**	library of object files
file.**d**	**tcov**(1) test coverage input file
file.**f**	Sun FORTRAN source file
file.**F**	Sun FORTRAN source file for **cpp**(1)
file.**for**	VMS FORTRAN source file for **f77cvt**(1)
file.**vf**	VMS FORTRAN source file for **f77cvt**(1)
file.**il**	**inline** expansion file
file.**o**	object file
file.**s**	assembler source file
file.**S**	assembler source for **cpp**(1)
file.**tcov**	output from **tcov**(1)
/lib/c2	optional optimizer for Sun-2, Sun-3, or Sun-4.
/lib/cg	Sun FORTRAN code generator

/lib/compile	compiler command-line processing driver
/lib/cpp	macro preprocessor
/lib/crt0.o	runtime startup
/lib/Fcrt1.o	startup code for **–fsoft** option
/lib/gcrt0.o	startup for gprof-profiling
/lib/libc.a	standard library, see **intro**(3)
/lib/mcrt0.o	startup for profiling
/lib/Mcrt1.o	startup code for **–f68881** option
/lib/optim	optional optimizer for Sun386*i*.
/lib/Scrt1.o	startup code for **–fsky** option
/lib/Wcrt1.o	startup code for **–ffpa** option
/usr/include/f77	directory searched by the Sun FORTRAN **INCLUDE** statement
/usr/bin/f77	compiler command-line processing driver
/usr/bin/f77cvt	VMS FORTRAN source code converter
/usr/lib/f77pass1	Sun FORTRAN parser
/usr/lib/libc_p.a	profiling library, see **intro**(3)
/usr/lib/libF77.a	Sun FORTRAN library: General - other than I/O or UNIX interface
/usr/lib/inline	inline expander of library calls
/usr/lib/libI77.a	Sun FORTRAN library: I/O routines
/usr/lib/libm.a	math library
/usr/lib/libpfc.a	startup code for combined Sun Pascal and Sun FORTRAN programs
/usr/lib/libU77.a	Sun FORTRAN library: interface to UNIX system calls
/tmp/*	compiler temporary files
mon.out	file produced for analysis by **prof**(1)
gmon.out	file produced for analysis by **gprof**(1)

SEE ALSO

cc(1), **f77cvt**(1), **fpr**(1), **fsplit**(1), **gprof**(1), **ld**(1), **prof**(1)

Sun FORTRAN Programmer's Guide

Floating-Point Programmer's Guide for the Sun Workstation

Programming Utilities and Libraries

DIAGNOSTICS

The diagnostics produced by **f77** itself are intended to be self-explanatory. Occasional messages may be produced by the linker.

A.7. ratfor Man Page

Action: After page 262 of the *Sun FORTRAN Programmer's Guide*, insert the page 262a that follows:

NAME

ratfor – rational FORTRAN dialect

SYNOPSIS

ratfor [**–6***c*] [**–C**] [**–h**] [filename ...]

DESCRIPTION

ratfor converts the rational FORTRAN dialect into standard FORTRAN 77. It provides control flow constructs essentially identical to those in C. See the *Sun FORTRAN Programmer's Guide* for a description of the Ratfor language.

OPTIONS

–6*c* Use the character *c* as the continuation character in column **6** when translating to FORTRAN. The default is to use the **&** character as a continuation character.

–C Pass Ratfor comments through to the translated code.

–h Translate Ratfor string constants to **H**ollerith constants of the form *nnn* **h** *string*. Otherwise just pass the strings through to the translated code.

SEE ALSO

f77(1)

Ratfor in the *Sun FORTRAN Programmer's Guide*

A.8. abort Man Page

Action: Remove pages 265 and 266 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 265 and 266 that follow.

Description of Changes

Removed all references to the optional argument `string`.

NAME

abort – terminate abruptly with memory image

SYNOPSIS

subroutine abort

DESCRIPTION

Abort cleans up the I/O buffers and then aborts producing a **core** file in the current directory.

FILES

/usr/lib/libF77.a

SEE ALSO

abort(3)

NAME

access – determine accessibility of a file

SYNOPSIS

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Access checks the given file, *name*, for accessability with respect to the caller according to *mode*. *Mode* may include in any order and in any combination one or more of:

r test for read permission

w test for write permission

x test for execute permission

(blank) test for existence

An error code is returned if either argument is illegal, or if the file can not be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libU77.a

SEE ALSO

access(2), perror(3F)

A.9. f77_ieee_environment Man Pages

Action: Remove pages 273 through 276 from the *Sun FORTRAN Programmer's Guide*, and replace them with the new pages 273 through 276 that follow.

Description of Changes

In the `f77_iee_environment` man pages, on page 275: inserted passing `sig` and `code` by *value*, using the `loc` function, and changed "`extern sample_handler`" to "`external sample_handler`".

NAME

f77_floatingpoint – Fortran IEEE floating-point definitions

SYNOPSIS

#include <f77/f77_floatingpoint.h>

DESCRIPTION

This file defines constants and types used to implement standard floating-point according to ANSI/IEEE Std 754-1985. Use these constants and types to write more easily understood .F source files that will undergo automatic preprocessing prior to Fortran compilation.

IEEE Rounding Modes:

fp_direction_type	The type of the IEEE rounding direction mode. Note that the order of enumeration varies according to hardware.
fp_precision_type	The type of the IEEE rounding precision mode, which only applies on systems that support extended precision such as Sun-3's with 68881's.

SIGFPE handling:

sigfpe_code_type	The type of a SIGFPE code.
sigfpe_handler_type	The type of a user-definable SIGFPE exception handler called to handle a particular SIGFPE code.
SIGFPE_DEFAULT	A macro indicating the default SIGFPE exception handling, namely for IEEE exceptions to continue with a default result, and to abort for other SIGFPE codes.
SIGFPE_IGNORE	A macro indicating an alternate SIGFPE exception handling, namely to ignore and continue execution.
SIGFPE_ABORT	A macro indicating an alternate SIGFPE exception handling, namely to abort with a core dump.

IEEE Exception Handling:

N_IEEE_EXCEPTION	The number of distinct IEEE floating-point exceptions.
fp_exception_type	The type of the N_IEEE_EXCEPTION exceptions. Each exception is given a bit number.
fp_exception_field_type	The type intended to hold at least N_IEEE_EXCEPTION bits corresponding to the IEEE exceptions numbered by *fp_exception_type*. Thus *fp_inexact* corresponds to the least significant bit and *fp_invalid* to the fifth least significant bit. Some operations may set more than one exception.

IEEE Classification:

fp_class_type	An enumeration of the various classes of IEEE floating-point values and symbols.

FILES

/usr/include/f77/f77_floatingpoint.h

SEE ALSO

ieee_environment(3M), f77_ieee_environment(3F)

NAME

IEEE environment - mode, status, and signal handling subprograms for IEEE arithmetic

SYNOPSIS

#include <f77/f77_floatingpoint.h>

integer function ieee_flags(action,mode,in,out)
character*(*) action, mode, in, out

integer function ieee_handler(action,exception,hdl)
character*(*) action, exception
sigfpe_handler_type hdl

sigfpe_handler_type function sigfpe(code, hdl)
sigfpe_code_type code
sigfpe_handler_type hdl

DESCRIPTION

These subprograms provide modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a FORTRAN program. They correspond closely to the functions *ieee_flags(3M)*, *ieee_handler(3M)*, and *sigfpe(3)*.

EXAMPLES

The following examples illustrate syntax.

```
integer ieeer
character*1 mode, out, in
ieeer = ieee_flags('clearall',mode, in, out)
```

sets ieeer to 0, rounding direction to 'nearest', rounding precision to 'extended', and all accrued exception-occurred status to zero.

```
character*1 out, in
ieeer = ieee_flags('clear','direction', in, out)
```

sets ieeer to 0, and rounding direction to 'nearest'.

```
character*1 out
ieeer = ieee_flags('set','direction','tozero',out)
```

sets ieeer to 0 and the rounding direction to 'tozero' unless the hardware does not support directed rounding modes; then ieeer is set to 1.

```
character*16 out
ieeer = ieee_flags('clear','exception','all',out)
```

sets ieeer to 0 and clears all accrued exception-occurred bits. If subsequently overflow, invalid, and inexact exceptions are generated then

```
character*16 out
ieeer = ieee_flags('get','exception','overflow',out)
```

sets ieeer to 25 and out to 'overflow'.

A user-specified signal handler might look like this:

```
      integer function sample_handler ( sig, code, sigcontext )
      integer sig
      integer code
      integer sigcontext(5)
c     Sample user-written sigfpe code handler.
c     Prints a message and terminates.
c     sig .eq. SIGFPE always.
c     The structure of sigcontext is defined in <signal.h>.
      print *, 'ieee exception code ', loc(code), ' occurred at pc ', sigcontext(4)
      call abort
      end
```

and it might be set up like this:

```
      external sample_handler
      integer ieeer
      ieeer = ieee_handler ( 'set', 'overflow', sample_handler )
      if (ieeer .ne. 0) print *,' ieee_handler can not set overflow '
```

NOTE: UNIX invokes signal handlers by passing **sig** and **code** by value. If you write a signal handler function in **FORTRAN** as in the example, you can access these values with the **loc** function.

FILES

/usr/include/f77/f77_floatingpoint.h
/usr/lib/libm.a

SEE ALSO

floatingpoint(3), signal(3), sigfpe(3), f77_floatingpoint(3F), ieee_flags(3M), ieee_handler(3M)

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*24 string

character*24 function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

Fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24  fdate
write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), idate(3F)

Systems for Open Computing™

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 37-29639

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor
Green Lane
Bagshot
Surrey, GU19 5NL
England
0276 51440
TLX 859017

Australia: (02) 436 4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
Nordic Countries: (08) 764 78 10
Switzerland: (1) 82 89 555
The Netherlands: 02155 24888
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276-51440

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales