


## *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 <code>OPEN(3S)</code> .	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
<pre>integer i real r external CSim i = 100 call CSim(i,r) ... ----- void csim_(int *i, float *r) {     *r = *i; }</pre>	<pre>int i=100; float r; extern void fsim_(int *i, float *r); fsim_(&amp;i, &amp;r); ... ----- subroutine FSim(i,r) integer i real r r = i return end</pre>

## 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
<pre>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 -&gt; r = 32.;     w -&gt; i = .007;     z -&gt; r = 66.67;     z -&gt; i = 94.1; }</pre>	<pre>struct cpx {float r, i;}; struct cpx d1; struct cpx *w = &amp;d1; struct dpx {double r, i;}; struct dpx d2; struct dpx *z = &amp;d2; fcmplx_( w, z ); ... ----- subroutine FCmplx( w, z ) complex w double complex z w = (32., .007) z = (66.67, 94.1) return end</pre>

## 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
<pre>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.;     ... }</pre>	<pre>extern void     qref_( int[][10], int *); ... int m[20][10] = ... ; int sum; ... qref_( m, &amp;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 ...</pre>

## 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	C calls Fortran
<pre> 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 -&gt; x;   v -&gt; x = v -&gt; y;   v -&gt; y = t;   v -&gt; z = -2.*(v -&gt; z); } </pre>	<pre> struct point {   float x,y,z; }; void fflip_ ( struct point *) ; ... struct point d; struct point *ptx = &amp;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 ... </pre>

Code Example 11-7 Passing Fortran 90 Derived Types

Fortran 90 calls C	C calls Fortran 90
<pre>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 -&gt; x;   v -&gt; x = v -&gt; y;   v -&gt; y = t;   v -&gt; z = -2.*(v -&gt; z); }</pre>	<pre>struct point {   float x,y,z; }; extern void fflip_ (   struct point *) ;  ... struct point d; struct point *ptx = &amp;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   ...</pre>

## 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
<pre> REAL X POINTER (P2X, X) EXTERNAL PASS P2X = MALLOC(4) X = 0. CALL PASS(X) ... ----- void pass_(x)   int **x; {   **x = 100.1; } </pre>	<pre> extern void fpass_; ... float *x; float **p2x; fpass_(p2x) ; ... ----- SUBROUTINE FPASS (P2X) REAL X POINTER (P2X, X) X = 0. ... </pre>

- 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 increments 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
<pre>REAL x, y x = 1. y = 0. PRINT *, x,y CALL value( %VAL(x), y) PRINT *, x,y END</pre>
<pre>----- void value_( float x, float *y) {     printf("%f, %f\n",x,*y);     x = x + 1.;     y = y + 1.;     printf("%f, %f\n",x,*y); } -----</pre>
<p>Compiling and running produces output:</p> <pre>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</pre>

## 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
<pre> real ADD1, R, S external ADD1 R = 8.0 S = ADD1( R ) ... ----- float add1_( pf ) float *pf; {     float f ;     f = *pf;     f++;     return ( f ); } </pre>	<pre> float r, s; extern float fadd1_() ; r = 8.0; s = fadd1_( &amp;r ); ... ----- real function fadd1 (p) real p fadd1 = p + 1.0 return end </pre>

## 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( <i>a1, a2, ..., an</i> )	cf_ (return, <i>a1, a2, ..., an</i> ) 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
<pre> COMPLEX U, V, RETCPX EXTERNAL RETCPX U = ( 7.0, -8.0) V = RETCPX(U) ... ----- struct complex { float r, i; }; void retcp_( temp, w ) struct complex *temp, *w; {     temp-&gt;r = w-&gt;r + 1.0;     temp-&gt;i = w-&gt;i + 1.0;     return; } </pre>	<pre> struct complex { float r, i; }; struct complex c1, c2; struct complex *u=&amp;c1, *v=&amp;c2; extern retfpx_(); u -&gt; r = 7.0; u -&gt; i = -8.0; retfpx_( v, u ); ... ----- COMPLEX FUNCTION RETFPX(Z) COMPLEX Z RETFPX = Z + (1.0, 1.0) RETURN END </pre>

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)	<pre> void c_ (result, length, a1, ..., an) char result[ ]; long length; </pre>

Here is an example:

Code Example 11-12 Function Returning CHARACTER String

Fortran calls C	C calls Fortran
<pre> 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&lt;count; i++) {     *cp++ = *p2arg ;   } } </pre>	<pre> 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,         &amp;ch, &amp;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 </pre>

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 ...	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
<pre>int altret_ ( int * ); main () {     int k, m ;     k =0;     m = altret_( &amp;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</pre>	<pre>demo% cc -c tst.c demo% f77 -o alt alt.f tst.o alt.f:     altret: demo% alt 1 2</pre> <p><i>The C routine receives the return value 2 from the Fortran routine because it executed the RETURN 2 statement.</i></p>