Chapter 11

C-Fortran Interface

This chapter treats issues regarding Fortran and C interoperability.

The discussion is inherently limited to the specifics of the Sun FORTRAN 77, Fortran 95, and C compilers.

Note – Material common to both FORTRAN 77 and Fortran 95 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. Depending on the situation, the choice is important:

- 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.

When a Fortran routine 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.

When a C function 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, and alternate returns are declared on `SUBROUTINE` statement, a zero is returned.

Data Type Compatibility

The tables below summarize the data sizes and default alignments for FORTRAN 77 and Fortran 95 data types. In both tables, note the following:

- C data types `int`, `long int`, and `long` are equivalent (4 bytes). In a 64-bit environment and compiling with `-xarch=v9` or `v9a`, `long` and pointers are 8 bytes. This is referred to as "LP64".
- `REAL*16` and `COMPLEX*32`, (`REAL(KIND=16)` and `COMPLEX(KIND=16)`), are available only on SPARC platforms. In a 64-bit environment and compiling with `-xarch=v9` or `v9a`, alignment is on 16-byte boundaries.
- Alignments marked 4/8 for SPARC indicate that alignment is 8-bytes by default, but on 4-byte boundaries in COMMON blocks. The maximum alignment in COMMON is 4-bytes.
- 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 `%VAL()` is not allowed in a Fortran dummy argument list.

FORTRAN 77 and C Data Types

TABLE 11-1 shows the sizes and allowable alignments for FORTRAN 77 data types. It assumes no compilation options affecting alignment or promoting default data sizes are applied. (See also the *FORTRAN 77 Language Reference Manual*).

TABLE 11-1 Data Sizes and Alignments--(in Bytes) Pass by Reference (`f77` and `cc`)

FORTRAN 77 Data Type	C Data Type	Size	Default Alignment SPARC	x86

`BYTE X`
`CHARACTER X`
`CHARACTER*`*n* X
	`char x`
	`unsigned char x`
	`unsigned char x[n]`
		1
		1
		n
			1
			1
			1
				1
				1
				1

`COMPLEX X`
`COMPLEX*8 X`
`DOUBLE COMPLEX X`
`COMPLEX*16 X`
`COMPLEX*32 X`
	`struct {float r,i;} x;`
	`struct {float r,i;} x;`
	`struct {double dr,di;}x;`
	`struct {double dr,di;}x;`
	`struct {long double dr,di;} x;`
		8
		8
		16
		16
		32
			4
			4
			4/8
			4/8
			4/8/16
				4
				4
				4
				4
				--

`DOUBLE PRECISION X`
`REAL X`
`REAL*4 X`
`REAL*8 X`
`REAL*16 X`
	`double x`
	`float x`
	`float x`
	`double x`
	`long double x`
		8
		4
		4
		8
		16
			4/8
			4
			4
			4/8
			4/8/16
				4
				4
				4
				4
				--

`INTEGER X`
`INTEGER*2 X`
`INTEGER*4 X`
`INTEGER*8 X`
	`int x`
	`short x`
	`int x`
	`long long int x`
		4
		2
		4
		8
			4
			2
			4
			4
				4
				2
				4
				4

`LOGICAL X`
`LOGICAL*1 X`
`LOGICAL*2 X`
`LOGICAL*4 X`
`LOGICAL*8 X`
	`int x`
	`char x`
	`short x`
	`int x`
	`long long int x`
		4
		1
		2
		4
		8
			4
			1
			2
			4
			4
				4
				1
				2
				4
				4

SPARC: Fortran 95 and C Data Types

The following table similarly compares the Fortran 95 data types with C.

TABLE 11-2 Data Sizes and Alignment--(in Bytes) Pass by Reference (`f95` and `cc`)

Fortran 95 Data Type	C Data Type	Size	Alignment
`CHARACTER x`	`unsigned char x ;`	1	1
`CHARACTER (LEN=n) x`	`unsigned char x[`*n*`] ;`	*n*	1
`COMPLEX x`	`struct {float r,i;} x;`	8	4
`COMPLEX (KIND=4) x` `COMPLEX (KIND=8) x`	`struct {float r,i;} x;` `struct {double dr,di;} x;`	8 16	4 4/8

COMPLEX (KIND=16) x	struct {long double, dr,di;} x;	32	4/8/16
DOUBLE COMPLEX x	struct {double dr, di;} x;	16	4/8
DOUBLE PRECISION x	double x ;	8	4
REAL x	float x ;	4	4
REAL (KIND=4) x REAL (KIND=8) x REAL (KIND=16) x	float x ; double x ; long double x ;	4 8 16	4 4/8 4/8/16
INTEGER x	int x ;	4	4
INTEGER (KIND=1) x INTEGER (KIND=2) x INTEGER (KIND=4) x INTEGER (KIND=8) x	signed char x ; short x ; int x ; long long int x;	1 2 4 8	4 4 4 4
LOGICAL x	int x ;	4	4
LOGICAL (KIND=1) x LOGICAL (KIND=2) x LOGICAL (KIND=4) x LOGICAL (KIND=8) x	signed char x ; short x ; int x ; long long int x;	1 2 4 8	4 4 4 4

Case Sensitivity

C and Fortran take opposite perspectives on case sensitivity:

- C is case sensitive--case matters.
- Fortran ignores case.

The `f77` and `f95` 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 Fortran program with the `-U` option, which tells the compiler 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 `f95` or `f77 -U` compiler option.

Underscores in Routine Names

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. 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 `C()` pragma to tell the Fortran compiler to omit those trailing underscores.
- Use the `f77` and `f95 -ext_names` option to compile references to external names without underscores.

Use only one of these solutions.

The examples in this chapter could use the `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. (Pragma directives are described in the *Fortran User's Guide*.)

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` and `f95` 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`
`      INTEGER B(3)`
`       ...`
`      CALL SAM( S, B(2) )`
	`      char s[7];`
	`      int b[3];`
	`        ...`
	`      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 preceding 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(1,1)  A(2,1)  A(3,1)  A(1,2)  A(2,2)  A(3,2)
```

C arrays in row-major order: `A[3][2]`

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

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 might be necessary.

For example, it may be confusing to do part of a matrix manipulation in C and the rest in Fortran. It might 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 in the following table.

TABLE 11-3 Comparing I/O Between Fortran and C

	Fortran Units	Standard I/O File Pointers	File Descriptors
Files Open	Opened for reading and writing	Opened for reading; or Opened for writing; or Opened for both; or Opened for appending; See `open`(2)	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 from 0-2147483647	Pointers to structures in the user's address space	Integers from 0-1023

File Permissions

C programs typically open input files for reading and output files for writing or for reading and writing. A `f77` program can `OPEN` a file `READONLY` or with `READWRITE='READ'` or `'WRITE'` or `'READWRITE'`. `f95` supports the `READWRITE` specifier, but not `READONLY`.

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 operation 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 `f95` Command

To link the proper Fortran and C libraries, use the `f77` or `f95` command to invoke the linker.

Example 1: Use `f77` to link:

```
demo% cc -c someCroutine.c

demo% f95 theF95routine.f someCroutine.o  ← The linking step

demo% a.out

 4.0 4.5

 8.0 9.0

demo%
```

Fortran Initialization Routines

Main programs compiled by `f77` and `f95` call dummy initialization routines `f77_init` or `f90_init` in the library at program start up. The routines in the library are dummies that do nothing. The calls the compilers generate pass pointers to the program's arguments and environment. These calls provide software hooks the programmer can use to supply their own routines, in C, to initialize their program in any customized manner before the program starts up.

One possible use of these initialization routines to call `setlocale` for an internationalized Fortran program. Because `setlocale` does not work if `libc` is statically linked, only Fortran programs that are dynamically linked with `libc` should be internationalized.

The source code for the `init` routines in the library is

```
void f77_init(int *argc_ptr, char ***argv_ptr, char ***envp_ptr) {}

void f90_init(int *argc_ptr, char ***argv_ptr, Char ***envp_ptr) {}
```

The routine `f77_init` is called by `f77` main programs. The routine `f90_init` is called by `f95` main programs. The arguments are set to the address of `argc`, the address of `argv`, and the address of `envp`.

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 and functions 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:

TABLE 11-4 Passing Simple Data Types

Fortran calls C	C calls Fortran

integer i
real r
external CSim
i = 100
call CSim(i,r)
...
---------- 	
void csim_(int *i, float *r)
{
*r = *i;
}
	int i=100;
	float r;
	extern void fsim_(int *i, float *r);
	fsim_(&i, &r);
	...
	---------- 	
	subroutine FSim(i,r)
	integer i
	real r
	r = i
	return
	end

COMPLEX Data

Pass a Fortran COMPLEX data item as a pointer to a C struct of two float or two double data types:

TABLE 11-5 Passing COMPLEX Data Types

Fortran calls C	C calls Fortran

`complex w`
`double complex z`
`external CCmplx`
`call CCmplx(w,z)`
`...`
`------------------------------`
`struct cpx {float r, i;};`
`struct dpx {double r,i;};`
`void ccmplx_(`
`struct cpx *w,`
`struct dpx *z)`
`{`
`w -> r = 32.;`
`w -> i = .007;`
`z -> r = 66.67;`
`z -> i = 94.1;`
`}`
	`struct cpx {float r, i;};`
	`struct cpx d1;`
	`struct cpx *w = &d1;`
	`struct dpx {double r, i;};`
	`struct dpx d2;`
	`struct dpx *z = &d2;`
	`fcmplx_( w, z );`
	`...`
	`---------------`
	`subroutine FCmplx( w, z )`
	`complex w`
	`double complex z`
	`w = (32., .007)`
	`z = (66.67, 94.1)`
	`return`
	`end`

In 64-bit environments and compiling with `-xarch=v9`, `COMPLEX` values are returned in registers.

Character Strings

Passing strings between C and Fortran routines is not recommended because there is no standard interface. However, note the following:

- All C strings are passed by reference.
- Fortran calls pass an additional argument for every argument with 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 appear after the explicit arguments in the call.

A Fortran call with a character string argument is shown in the next example with its C equivalent:

TABLE 11-6 Passing a CHARACTER string

Fortran call:	C equivalent:

`CHARACTER*7 S`
`INTEGER B(3)`
`...`
`CALL CSTRNG( S, B(2) )`
`...`
	`char s[7];`
	`int 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.

TABLE 11-7 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 )`
	`extern void vecref_`
	`( int[], int * );`
	`...`
	`int i, sum;`
	`int v[9] = ...`
	`vecref_( v, &sum );`
	`...`
	`------------------------------`
	`subroutine VecRef( v, total)`

```
{
int i;
*sum = 0;
for ( i = 0; i <= 8; i++ )
*sum = *sum + v[i];
}
```

```
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.

TABLE 11-8 Passing a Two-Dimensional Array

Fortran calls C	C calls Fortran

REAL Q(10,20)
...
Q(3,5) = 1.0
CALL FIXQ(Q)
...
------------------------------ 	
void fixq_(float a[20][10])
{
...
a[5][3] = a[5][3] + 1.;
...
}
	extern void
	qref_(int[][10], int *);
	...
	int m[20][10] = ... ;
	int sum;
	...
	qref_(m, &sum);
	...
	------------------------------ 	
	SUBROUTINE QREF(A,TOTAL)
	INTEGER A(10,20), TOTAL
	DO I = 1,10
	DO J = 1,20
	TOTAL = TOTAL + A(I,J)
	END DO
	END DO
	...

Structures

C and FORTRAN 77 structures and Fortran 95 derived types can be passed to each other's routines as long as the corresponding elements are compatible.

TABLE 11-9 Passing FORTRAN 77 STRUCTURE Records

Fortran calls C	C calls Fortran

STRUCTURE /POINT/
REAL X, Y, Z
END STRUCTURE
RECORD /POINT/ BASE
EXTERNAL FLIP
...
CALL FLIP(BASE)
...
------------------------------ 	
struct point {
float x,y,z;
};
void flip_(struct point *v)
{
float t;
t = v -> x;
v -> x = v -> y;
v -> y = t;
v -> z = -2.*(v -> z);
}
	struct point {
	float x,y,z;
	};
	void fflip_ (struct point *) ;
	...
	struct point d;
	struct point *ptx = &d;
	...
	fflip_ (ptx);
	...
	------------------------------ 	
	SUBROUTINE FFLIP(P)
	STRUCTURE /POINT/
	REAL X,Y,Z
	END STRUCTURE
	RECORD /POINT/ P
	REAL T
	T = P.X
	P.X = P.Y
	P.Y = T
	P.Z = -2.*P.Z
	...

TABLE 11-10 Passing Fortran 95 Derived Types

Fortran 95 calls C	C calls Fortran 95
TYPE point SEQUENCE	struct point { float x,y,z;

```
REAL :: x, y, z
END TYPE point
TYPE (point) base
EXTERNAL flip
...
CALL flip( base)
...
-----------------------------
struct point {
float x,y,z;
};
void flip_( struct point *v )
{
float t;
t = v -> x;
v -> x = v -> y;
v -> y = t;
v -> z = -2.*(v -> z);
}
```

```
};
extern void fflip_ (
struct point *) ;
...
struct point d;
struct point *ptx = &d;
...
fflip_ (ptx);
...
-----------------------------
SUBROUTINE FFLIP( P )
TYPE POINT
REAL :: X, Y, Z
END TYPE POINT
TYPE (POINT) P
REAL :: T
T = P%X
P%X = P%Y
P%Y = T
P%Z = -2.*P%Z
...
```

Pointers

A FORTRAN 77 pointer can be passed to a C routine as a pointer to a pointer because the Fortran routine passes arguments by reference.

TABLE 11-11 Passing a FORTRAN 77 POINTER

Fortran calls C	C calls Fortran

REAL X
POINTER (P2X, X)
EXTERNAL PASS
P2X = MALLOC64(4)
X = 0.
CALL PASS(P2X)
...
---------- 	
void pass_(x)
int **x;
{
**x = 100.1;
}
	extern void fpass_(p2x);
	...
	float *x;
	float *p2x;
	p2x = &x;
	fpass_(p2x) ;
	...
	---------- 	
	SUBROUTINE FPASS (P2X)
	REAL X
	POINTER (P2X, X)
	X = 0.
	...

C pointers are compatible with Fortran 95 *scalar* pointers, but not *array* pointers.

Passing Data Arguments by Value

Call by value is available only 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 function `%VAL(`*arg*`)` as an argument in the call.

In the following example, the Fortran routine passes x by value and y by reference. The C routine incremented both x and y, but only y is changed.

TABLE 11-12 Passing Simple Data Arguments by Value: FORTRAN 77 Calling C

Fortran 77 calls C
REAL x, y x = 1. y = 0. PRINT *, x,y CALL value(%VAL(x), y) PRINT *, x,y END

```
------------------------------------------------------
void value_( float x, float *y)
{
printf("%f, %f\n",x,*y);
x = x + 1.;
*y = *y + 1.;
printf("%f, %f\n",x,*y);
}
------------------------------------------------------
```

Compiling and running produces output:

```
1.00000 0.  x and y from Fortran
1.000000, 0.000000  x and y from C
2.000000, 1.000000  new x and y from C
1.00000 1.00000  new x and y from Fortran
```

Functions That Return a Value

A Fortran function that returns a value of type BYTE (*FORTRAN 77 only*), INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or REAL*16 (*SPARC 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.

Returning a 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:

TABLE 11-13 Functions Returning a REAL or float Value

Fortran calls C	C calls Fortran

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

Returning COMPLEX Data

A Fortran function returning COMPLEX or DOUBLE COMPLEX on SPARC V8 platforms is equivalent to a C function with an additional first argument that points to the return value in memory. The general pattern for the Fortran function and its corresponding C function is:

Fortran function	C function
`COMPLEX FUNCTION CF(`*a1, a2, ..., an*`)`	`cf_ (`*return, a1, a2, ..., an*`)` `struct { float r, i; } *`*return*

TABLE 11-14 Function Returning COMPLEX Data

Fortran calls C	C calls Fortran

`COMPLEX U, V, RETCPX`
`EXTERNAL RETCPX`
`U = ( 7.0, -8.0)`
`V = RETCPX(U)`
	`struct complex { float r, i; };`
	`struct complex c1, c2;`
	`struct complex *u=&c1, *v=&c2;`
	`extern retfpx_();`

```
...
------------------------------
struct complex { float r, i; };
void retcpx_( temp, w )
struct complex *temp, *w;
{
temp->r = w->r + 1.0;
temp->i = w->i + 1.0;
return;
}
```

```
u -> r = 7.0;
u -> i = -8.0;
retfpx_( v, u );
...
------------------------------
COMPLEX FUNCTION RETFPX(Z)
COMPLEX Z
RETFPX = Z + (1.0, 1.0)
RETURN
END
```

In 64-bit environments and compiling with `-xarch=v9`, `COMPLEX` values are returned in floating-point registers: `COMPLEX` and `DOUBLE COMPLEX` in `%f0` and `%f1`, and `COMPLEX*32` in `%f0`, `%f1`, `%f2`, and `%f3`. These registers are not directly accessible to C programs, preventing such interoperability between Fortran and C on SPARC V9 platforms for this case.

Returning a CHARACTER String

Passing strings between C and Fortran routines is not encouraged. However, a Fortran character-string-valued function is equivalent to a C function with two additional first arguments--data address and string length. The general pattern for the Fortran function and its corresponding C function is:

Fortran function	C function

`CHARACTER*`*n* `FUNCTION C(`*a1, ..., an*`)`	`void c_ (`*result, length, a1, ..., an*`)`
	`char` *result*`[ ];`
	`long length;`

Here is an example:

TABLE 11-15 A Function Returning a CHARACTER String

Fortran calls C	C calls Fortran

`CHARACTER STRING*16, CSTR*9`
`STRING = ' '`
`STRING = '123' // CSTR('*',9)`
`...`
`------------------------------`
`void cstr_( char *p2rslt,`
`long rslt_len,`
`char *p2arg,`
`long *p2n,`
`long arg_len )`
`{ /* return n copies of arg */`
`int count, i;`
`char *cp;`
`count = *p2n;`
`cp = p2rslt;`
`for (i=0; i<count; i++) {`
`*cp++ = *p2arg ;`
`}`
`}`
	`void fstr_( char *, long,`
	`char *, long *, long );`
	`char sbf[9] = "123456789";`
	`char *p2rslt = sbf;`
	`int rslt_len = sizeof(sbf);`
	`char ch = '*';`
	`int n = 4;`
	`int ch_len = sizeof(ch);`
	`/* make n copies of ch in sbf`
	`*/`
	`fstr_( p2rslt, rslt_len,`
	`&ch, &n, ch_len );`
	`...`
	`------------------------------`
	`FUNCTION FSTR( C, N)`
	`CHARACTER FSTR*(*), C`
	`FSTR = ''`
	`DO I = 1,N`
	`FSTR(I:I) = C`
	`END DO`
	`FSTR(N+1:N+1) = CHAR(0)`
	`END`

In this example, the C function and calling C routine must accommodate two initial extra arguments (a pointer to the result string and the length of the 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`.

TABLE 11-16 Emulating 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 an underscore to link with the block created by the Fortran program. Note also that the C directive `#pragma pack` may be needed to get the same padding as with Fortran. Both `f77` and `f95` align data in COMMON blocks to at most 4-byte boundaries.

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, 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 implementation dependent and its use should be avoided.

TABLE 11-17 Alternate Returns

C calls Fortran	Running the Example

int altret_ (int *);
main ()
{
int k, m ;
k =0;
m = altret_(&k) ;
printf("%d %d\n", k, m);
	demo% **cc -c tst.c**
	demo% **f77 -o alt alt.f tst.o**
	alt.f:
	altret:
	demo% **alt**
	1 2

`}`
`------------------------------`
`SUBROUTINE ALTRET( I, *, *)`
`INTEGER I`
`I = I + 1`
`IF(I .EQ. 0) RETURN 1`
`IF(I .GT. 0) RETURN 2`
`RETURN`
`END`
	The C routine receives the return value 2 from the Fortran routine because it executed the RETURN 2 statement.