Chapter 3

Fortran Program Interfaces

This chapter contains the following major sections:

- "Fortran/C Interface" describes the interface between Fortran routines and routines written in C. It contains rules and gives examples for making calls and passing arguments between the two languages.
- "Fortran/C Wrapper Interface" describes the process of generating wrappers for C routines called by Fortran.
- "Fortran/Pascal Interface" describes the interface between Fortran routines and routines written in Pascal. It contains rules and gives examples for making calls and passing arguments between the two languages.

You may need to refer to other sources of information as you read this chapter.

- For information on storage mapping (how the variables of the various languages appear in storage), refer to Chapter 1 for Fortran and to Chapter 2 in the appropriate language programmer's guide for other languages.
- For information on the standard linkage conventions used by the compiler in generating code, see Chapter 7 of the *Assembly Language Programmer's Guide.*

For information on built–in functions that provide access to non–Fortran system functions and library routines, see Chapter 4 of this manual.

Fortran/C Interface

When writing Fortran programs that call C functions, consider procedure and function declaration conventions for both languages. Also, consider the rules for argument passing, array handling, and accessing common blocks of data.

Procedure and Function Declarations

This section discusses items to consider before calling C functions from Fortran.

Names

When calling a Fortran subprogram from C, the C program must append an underscore (_) to the name of the Fortran subprogram. For example, if the name of the subprogram is **matrix**, then call it by the name **matrix_**. When Fortran is calling a C function, the name of the C function must also end with an underscore.

The Fortran compiler changes all its subprogram names to lowercase. Thus, all of the following subprograms refer to the same function **matrix** when interfacing with C:

```
subroutine MATRIX
```

```
subroutine Matrix
subroutine matrix
```

The exception to this rule is when the **-u** option to *f77* is used. This option causes case to be preserved.

Note that only one main routine is allowed per program. The main routine can be written in either C or Fortran. Table 3–1contains an example of a C and a Fortran main routine.

Table 3–1 Main Routines

C	Fortran

main () {
printf("hi!\n");
}
	write (6,10)
	10 format ('hi!')
	end

Invocations

Invoke a Fortran subprogram as if it were an integer–valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the subprogram but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined. The Fortran statement

```
call nret (*1,*2Ex,*3)
```

is treated exactly as if it were the computed **goto**

```
goto (1,2,3), nret()
```

A C function that calls a Fortran subprogram can usually ignore the return value of a Fortran subroutine; however, the C function should not ignore the return value of a Fortran function. Table 3–2shows equivalent function and subprogram declarations in C and Fortran programs.

Table 3–2 Equivalent C and Fortran Function Declarations

C Function Declaration	Fortran Function Declaration
double dfort()	double precision function dfort()
double rfort()	real function rfort()
int ifort()	integer function ifort()
int lfort	logical function lfort()

Note the following:

- Avoid calling Fortran functions of type **FLOAT**, **COMPLEX**, and **CHARACTER** from C.
- You cannot write a C function so that it will return a **COMPLEX** value to Fortran.
- A character–valued Fortran subprogram is equivalent to a C language routine with two extra initial arguments: a data address and a length. However, if the length is one, no extra argument is needed and the single character result is returned as in a normal numeric function.

  Thus

  ```
  character*15 function g(...)
  ```

  is equivalent to

```
char result [1];
long int length;
g_(result, length, …)
…
```

and could be invoked in C by

```
char chars[15]
g_(chars, 15, …);
```

and

```
character function h(…)
```

could be invoked in C by

```
char c, h();
c=h_(…);
```

Arguments

The following rules apply to arguments passed between Fortran and C:

- All explicit arguments must be passed by reference. All routines must specify an *address* rather than a value. Thus, to pass constants or expressions to Fortran, the C routine must first store their values into variables and then pass the address of the variable. (The only exception occurs when passing the length of a string from C to a Fortran subroutine with a parameter of type **CHARACTER**.)
- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in Table 3–3

Table 3–3 Equivalent Fortran and C Data Types

Fortran	C
integer*2 x	short int x;
integer x	long int x; or just int x;
logical x	long int x; or just int x;
real x	float x;
double precision x	double x;
complex x	struct{float real, imag;) x;
double complex x	struct{double dreal,dimag;} x;
character*6 x	char x[6] [a]

[a] The array length must also be passed, as discussed in the next section.

- Note that in Fortran, **INTEGER** and **LOGICAL** variables occupy 32 bits of memory by default, but this can be changed by using the **-i2** option.
- The Fortran compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
  - destination address for character functions, when called

– length of a character variable, when an argument is the address of a character variable

When a C function calls a Fortran routine, the C function must explicitly specify these items in its argument list *in the following order:*

1. If the Fortran routine is a function that returns a character variable of length greater than 1, specify the address and length of the resultant character variable.
2. Specify normal arguments (addresses of arguments or functions).
3. Specify the length of each normal character parameter in the order it appeared in the argument list. The length must be specified as a constant value or **INTEGER** variable (that is, not an address).

The examples on the following pages illustrate these rules.

Example 1

This example shows how a C routine specifies the destination address of a Fortran function (which is only implied in a Fortran program).

Fortran

```
C  Fortran call to SAM, a routine written
C  in Fortran
   EXTERNAL F
   CHARACTER*7 S
   INTEGER B(3)
   …
   CALL SAM (F, B(2), S)
```

C

```
/* C call to SAM, a routine written in Fortran */
/* We pass in the function pointer for the     */
/* Fortran SUBROUTINE F                        */
char s[7];
int  b[3];
extern void sam_(void (*)(), int *, char*);
                        /* Fortran subroutine SAM          */
extern void f_();       /* Fortran subroutine F            */
…
sam_(F, &B[1], S);      /* We pass in pointer to Fortran F */
                        /* for Fortran call-by-reference   */
```

Example 2

This example shows how a C routine must specify the length of a character string (which is only implied in a Fortran call).

Fortran

```
C  Fortran call to F, a function written
C  in Fortran
   EXTERNAL F
   CHARACTER*10 F, G
   G = F()
```

C

```
/* C call to SAM, a routine written in Fortran */
/* which returns a string.                     */
CHAR S[10];
. . .
f_(S, 10);
```

The function F, written in Fortran

```
C  function F, written in Fortran
   CHARACTER*10 FUNCTION F()
   F = ‘0123456789’
   RETURN
   END
```

Array Handling

Fortran stores arrays in column−major order with the leftmost subscript varying the fastest. C, however, stores arrays in the opposite arrangement (row−major order), with the rightmost subscripts varying the fastest.

Here is how the layout of the Fortran array looks:

```
integer t (2,3)
t(1,1), t(2,1), t(1,2), t(2,2), t(1,3), t(2,3)
```

Here is how the layout of the C array looks:

```
int t [2] [3]
t[0][0], t[0][1], t[0][2], t[1][0], t[1][0], t[1][1],t[1][2]
```

Note that the default for the lower bound of an array in Fortran is 1, where the default in C is 0.

When a C routine uses an array passed by a Fortran subprogram, the dimensions of the array and the use of the subscripts must be interchanged, as shown in Figure 3−1

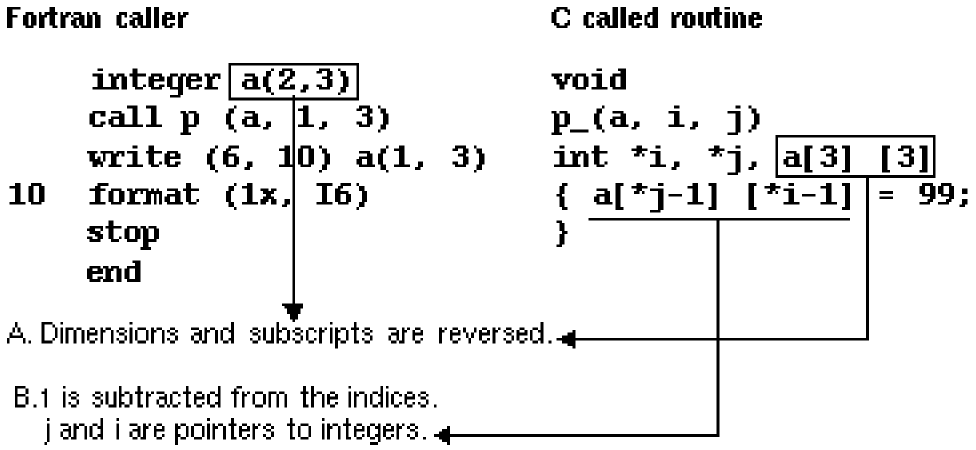

Figure 3–1 Array Subscripts

The Fortran caller prints out the value 99. Note the following:

- Because arrays are stored in column–major order in Fortran and row– major order in C, the dimension and subscript specifications are reversed.
- Because the lower–bound default is 1 for Fortran and 0 for C, 1 must be subtracted from the indexes in the C routine. Also, because Fortran passes parameters by reference, ***j** and ***p** are pointers in the C routine.

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- Fortran common blocks must be declared by common statements; C can use any global variable. Note that the common block name in C (**sam_**) must end with an underscore.
- Data types in Fortran and C programs must match unless you want equivalencing. If so, you must adhere to the alignment restrictions for the data types described in Chapter 2.
- If the same common block is of unequal length, the largest size is used to allocate space.
- Unnamed common blocks are given the name **_BLNK_**.

The following examples show C and Fortran routines that access common blocks of data.

Fortran

```
subroutine sam()
common /r/ i, r
i = 786
r = 3.2
return
```

```
end
```

C

```
struct S {int i; float j;}r_;
main () {
sam_() ;
printf("%d %f\n",r_.i,r_.j);
}
```

The C routine prints out 786 and 3.2.

Fortran/C Wrapper Interface

This section describes the process of generating wrappers for C routines called by Fortran. If you want to call existing C routines (which use value parameters rather than reference parameters) from Fortran, these *wrappers* convert the parameters during the call.

The program *mkf2c* provides an alternate interface for C routines called by Fortran.

Fortran routines called by C must use the method described in "Fortran/C Wrapper Interface".

The Wrapper Generator mkf2c

The *mkf2c* program uses C data−type declarations for parameters to generate the correct assembly language interface. In generating a Fortran−callable entry point for an existing C−callable function, the C function is passed through *mkf2c*, and *mkf2c* adds additional entry points. Native language entry points are not altered.

Use these rules with *mkf2c*:

- Each function given to *mkf2c* must have the standard C function syntax.
- The function body must exist but can be empty. Function names are transformed as necessary in the output.

A simple case of using a function as input to *mkf2c* is

```
func()
{}
```

Here, the function **func** has no parameters. If *mkf2c* is used to produce a Fortran−to−C wrapper, the Fortran entry is **func_**. The wrapper **func_** simply calls the C routine **func***()*.

Here is another example:

```
simplefunc (a)
int a;
{}
```

In this example, the function **simplefunc** has one argument, **a**. The argument is of type **int**. For this function, **mkf2c** produces three items: a Fortran entry, **simple**, and two pieces of code. The first piece of code dereferences the address of **a**, which was passed by Fortran. The second passes the resulting **int** to C. It then calls the C routine **simplefunc()**.

Using Fortran Character Variables as Parameters

You can specify the length of a character variable passed as a parameter to Fortran either at compilation or at run time. The length is determined by the declaration of the parameter in the Fortran routine. If the declaration contains a length, the passed length must match the declaration. For example, in the following declaration, the length of the string is declared to be 10 characters:

```
character*10 string
```

The passed length must be 10 in order to match the declaration.

When this next declaration is used, the passed length is taken for operations performed on the variable inside the routine:

```
character*(*) string
```

The length can be retrieved by use of the Fortran intrinsic function **LEN**. Substring operations may cause Fortran run−time errors if they do not check this passed length.

Arrays of character variables are treated by Fortran as simple byte arrays, with no alignment of elements. The length of the individual elements is determined by the length passed at run time. For instance, the array **sarray()** can be declared in this manner:

```
character*(*) sarray()
```

This length is necessary to compute the indexes of the array elements. The program *mkf2c* has special constructs for dealing with the lengths of Fortran character variables.

Reduction of Parameters

The program *mkf2c* reduces each parameter to one of seven simple objects. The following list explains each object.

64−bit value — The quantity is loaded indirectly from the passed address, and the result is passed to C. Parameters with the C type double (or long float) are reduced to 64−bit values by converting the 32−bit Fortran **REAL** parameter to double precision (see below).

32−bit value — *mkf2c* uses the passed address to retrieve a 32−bit data value, which is passed to C. Parameters with C types **int** and **long** are reduced to 32−bit values. Any parameter with an unspecified type is assumed to be **int**. If the **-f** option is specified, parameters with the C type **float** are reduced to 32−bit values.

16−bit value — A 16−bit value is loaded using the passed address. The value is either extended (if type is **signed** in the function parameter list) or masked (if type is **unsigned**) and passed to C. Any parameter whose C type is **short** is reduced to a 16−bit value.

8−bit value — The **char** type in C corresponds to the **CHARACTER*1** type in Fortran 77. (There is no mechanism to pass **integer*1** variables to C. A pointer to the value can be passed by declaring the parameter as **int***.) By default the character value is loaded as an unsigned quantity and passed to C. If the **-signed** option has been specified when invoking *mkf2c*, the character value is sign extended before being passed to C.

character string A copy is made of the Fortran character variable, and it is null terminated, and passed as a character pointer to C. Any modifications that C makes to the string will not affect the corresponding character variable in the Fortran routine.

character array When using *mkf2c* to call C from Fortran, the address of the Fortran character variable is passed. This character array can be modified by C. It is not guaranteed to be null terminated. The length of the Fortran character variable is treated differently (as discussed in the next section).

pointer The value found on the stack is treated as a pointer and is passed without alteration. Any array or pointer that is not of type **char**, any parameter with multiple levels of indirection, or any indirect array is assumed to be of type **pointer**. If the type of a parameter is specified but is not one of the standard C types, *mkf2c* will pass it as a pointer.

Below is an example of a C specification for a function:

```
test (i,s,c,ptr1,ar1,u,f,d,d1,str1,str2,str3)
short s;
unsigned char c;
int *ptr1;
char *ptr2[];
short ar1[];
sometype u;
float f;
long float d, *d1;
char *str1;
char str2[],str3[30];
 {
   /* The C function body CAN go here. Nothing
   except the opening and closing braces are
   necessary */
```

If this function were passed to *mkf2c*, the parameters would be transformed as follows:

- **PTR1**, **PTR2**, **AR1**, **D1**, and **U** would be passed as simple pointers.
- *mkf2c* would complain about not understanding the type **SOMETYPE** but, by default, would assume it to be of type **POINTER**.
- **S**, **C**, and **D** would be passed as values of length 16 bits, 64 bits, and 8 bits, respectively. **F** would be converted to a 64–bit **DOUBLE** before being passed, unless the **-f** option had been specified. If the **-f** option had been specified, **F** would be passed as a 32–bit value. Because the type of **I** is not specified, it would be assumed to be **INT** and would also be passed as a 32–bit value. Storing values in any of these parameters would not have any effect on the original Fortran data.

Fortran Character Array Lengths

When the wrapper generator is used, a character variable that is specified as **char*** in the C parameter list is *copied* and null terminated. C may thus determine the length of the string by the use of the standard C function **strlen**.

If a character variable is specified as a character array in the C parameter list, the address of the character variable is passed, making it impossible for C to determine its length, as it is not null terminated. When the call occurs, the wrapper code receives this length from Fortran. For those C functions needing this information, the wrapper passes it by extending the C parameter list.

For example, if the C function header is specified as follows

```
func1 (carr1,i,str,j,carr2)
char carr1[],*str,carr2[];
int i, j;
{}
```

mkf2c will pass a total of seven parameters to C. The sixth parameter will be the length of the Fortran character variable corresponding to **carr1**, and the seventh will be the length of **carr2**. The C function **func1()** must use the **varargs** macros to retrieve these hidden parameters. *mkf2c* will ignore the

varargs macro **va_alist** appearing at the end of the parameter name list and its counterpart **va_alist** appearing at the end of the parameter type list. In the case above, use of these macros would produce the function header

```
#include "varargs.h"
func1 (carr1,i,str,j,carr2,va_alist)
char carr1[], *str, carr2[];
int i, j;
va_dcl
{}
```

The C routine could retrieve the lengths of **carr1** and **carr2**, placing them in the local variables **carr1_len** and **carr2_len** by the following code fragment:

```
va_list ap;
int carr1_len, carr2_len;
va_start(ap);
carr1_len = va_arg (ap, int)
carr2_len = va_arg (ap, int)
```

Using mkf2c and extcentry

mkf2c understands only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C−style comments, and function bodies. However, it cannot understand constructs such as typedefs, external function declarations, or C preprocessor directives.

To ensure that only those constructs understood by *mkf2c* are included in wrapper input, you need to place special comments around each function for which Fortran–to–C wrappers are to be generated (see example below).

Once these special comments, **/* CENTRY */** and **/* ENDCENTRY */**, are placed around the code, use the program *excentry*(1) before *mkf2c* to generate the input file for *mkf2c.*

To illustrate the use of *extcentry*, the C file *foo.c* is shown below. It contains the function *foo*, which is to be made Fortran callable.

```
typedef unsigned short grunt [4];
struct {
   long 1,11;
   char *str;
} bar;
main ()
{
   int kappa =7;
   foo (kappa,bar.str);
}
/* CENTRY */
foo (integer, cstring)
int integer;
char *cstring;
{
   if (integer==1) printf("%s",cstring);
} /* ENDCENTRY */
```

The special comments **/* CENTRY */** and **/* ENDCENTRY */** surround the section that is to be made Fortran callable. To generate the assembly language wrapper **foowrp.s** from the above file **foo.c**, use the following set of commands:

```
% extcentry foo.c foowrp.fc% mkf2c foowrp.fc foowrp.s
```

The programs *mkf2c* and *extcentry* are found in the directory */usr/bin* on your workstation.

Makefile Considerations

make(1) contains default rules to help automate the control of wrapper generation. The following example of a makefile illustrates the use of these rules. In the example, an executable object file is created from the files **main.f** (a Fortran main program) and **callc.c**:

```
test:  main.o callc.o
   f77 –o test main.o callc.o
callc.o: callc.fc
clean:
   rm –f *.o test *.fc
```

In this program, **main** calls a C routine in **callc.c**. The extension **.fc** has been adopted for Fortran–to–call–C wrapper source files. The wrappers created from **callc.fc** will be assembled and combined with the binary created from **callc.c**. Also, the dependency of **callc.o** on **callc.fc** will cause **callc.fc** to be recreated from **callc.c** whenever the C source file changes. (The programmer is responsible for placing the special comments for *extcentry* in the C source as required.)

Note: Options to *mkf2c* can be specified when *make* is invoked by setting the *make* variable **F2CFLAGS**. Also, do not create a **.fc** file for the modules that need wrappers created. These files are both created and removed by *make* in response to the **file.o:file.fc** dependency.

The *makefile* above will control the generation of wrappers and Fortran objects. You can add modules to the executable object file in one of the following ways:

- If the file is a native C file whose routines are not to be called from Fortran using a wrapper interface, or if it is a native Fortran file, add the **.o** specification of the final *make* target and dependencies.
- If the file is a C file containing routines to be called from Fortran using a wrapper interface, the comments for *extcentry* must be placed in the C source, and the **.o** file placed in the target list. In addition, the dependency of the **.o** file on the **.fc** file must be placed in the *makefile.* This dependency is illustrated in the example *makefile* above where **callf.o** depends on **callf.fc**.

Fortran/Pascal Interface

This section discusses items you should consider when writing a call between Fortran and Pascal.

Procedure and Function Declarations

This section explains procedure and function declaration considerations.

Names

In calling a Fortran program from Pascal, you must place an underscore (_) as a suffix to routine names and data names.

To call Fortran from Pascal or vice versa, specify an underscore (_) as the suffix of the name of the Fortran or Pascal routine being called. For example, if the routine is called *matrix*, then call it by the name **matrix_**.

In Pascal, always declare the external Fortran subprogram or function with **VAR** parameters.

Note that only one main routine is allowed per program. The main routine can be written either in Pascal or Fortran. Table 3−4contains an example of a Pascal and a Fortran main routine.

Table 3−4 Main Routines

Pascal	Fortran
program p;	write (6,10)
begin	10 format (’hi!’)
writeln (’hi!’);	stop
end.	end

Invocation

If you have alternate return labels, you can invoke a Fortran subprogram as if it were an integer−valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined.

The Fortran statement

```
call nret (*1,*2,*3)
```

is treated exactly as if it were the computed *goto*

```
goto (1,2,3), nret()
```

A Pascal function that calls a Fortran subroutine can usually ignore the return value. Table 3−5shows equivalent function declarations in Pascal and Fortran.

Table 3−5 Function Declarations

Pascal	Fortran
function dfort_(): double;	double precision function dfort()
function rfort_(): real;	real function rfort()
function ifort_(): integer;	integer function ifort()

Fortran has a built−in data type**COMPLEX** that does not exist in Pascal. Therefore, there is no compatible way of returning these values from Pascal.

A character−valued Fortran function is equivalent to a Pascal language routine with two initial extra arguments: a data address and a length.

The following Fortran statement

```
character*15 function g (...)
```

is equivalent to the Pascal code

```
type string = array [1..15];
var
   length: integer;
   a: array[1..15] of char;
procedure g_(var a:string;length:integer;...); external;
```

and could be invoked by the Pascal line

```
g_ (a, 15);
```

Arguments

The following rules apply to argument specifications in both Fortran and Pascal programs:

- All arguments must be passed by reference. That is, the argument must specify an *address* rather than a value. Thus, to pass constants or expressions, their values must first be stored into variables and then the addresses of the variables passed.

- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in Table 3–6

Table 3–6 Equivalent Fortran and Pascal Data Types

Pascal	Fortran
integer	integer*4, integer, logical
cardinal, char, boolean, enumeration	character
real	real
double	double precision
procedure	subroutine
record r:real; i:real; end;	complex
record r:double; i:double; end;	double complex

- Note that Fortran requires that each **INTEGER**, **LOGICAL**, and **REAL** variable occupy 32 bits of memory.
- Functions of type **INTEGER**, **REAL**, or **DOUBLE PRECISION** are interchangeable between Fortran and Pascal and require no special considerations.
- The Fortran compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
  - destination address for character functions, when called
  - length of character strings, when an argument is the address of a character string

When a Pascal program calls a Fortran subprogram, the Pascal program must explicitly specify these items in its argument list *in the following order*:

1. Destination address of character function.
2. Normal arguments (addresses of arguments or functions).
3. Length of character strings. The length must be specified as an absolute value or **INTEGER** variable. The next two examples illustrate these rules.

Example

The following example shows how a Pascal routine must specify the length of a character string (which is only implied in a Fortran call).

Fortran call to SAM

```
C  SAM IS A ROUTINE WRITTEN IN FORTRAN
```

```
   EXTERNAL F
   CHARACTER*7 S
   INTEGER B(3)
   …
   CALL SAM (F, B(1), S) <- Length of S is implicit.
```

Pascal call to SAM

```
PROCEDURE F_; EXTERNAL;
S: ARRAY[1..7] OF CHAR;
B: ARRAY[1..3] OF INTEGER;
   …
SAM_ (F, B[1], S, 7);  <- Length of S is explicit.
```

Execution−Time Considerations

Pascal checks certain variables for errors at execution time, whereas Fortran does not. For example, in a Pascal program, when a reference to an array exceeds its bounds, the error is flagged (if run−time checks are not suppressed). Use the **f77-c** option if you want a Fortran program to detect similar errors when you pass data to it from a Pascal program.

Array Handling

Fortran stores arrays in column−major order, where the leftmost subscripts vary the fastest. Pascal, however, stores arrays in row−major order, with the rightmost subscript varying the fastest. Also, the default lower bound for arrays in Fortran is 1. Pascal has no default; the lower bound must be explicitly specified. Here is an example of the various layouts:

Fortran

```
integer t (2,3)
t(1,1), t(2,1), t(1,2), t(2,2), t(1,3), t(2,3)
```

Pascal

```
var t: array[1..2,1..3] of integer;
t[1,1], t[1,2], t[1,3], t[2,1], t[2,2], t[2,3]
```

When a Pascal routine uses an array passed by a Fortran program, the dimensions of the array and the use of the subscripts must be interchanged. The example below shows the Pascal code that interchanges the subscripts.

In the following example, the Fortran routine calls the Pascal procedure **p**, receives the value 99, and prints it out.

Fortran

```
   INTEGER A(2,3)
   CALL P (A, 1, 3)
   WRITE (6,10) A(1,3)
10 FORMAT (1X, I9)
```

```
   STOP
   END
```

Pascal

```
TYPE ARRY = ARRAY [1..3,1..2];
PROCEDURE P_(VAR A:ARRY; VAR I,J:INTEGER);
BEGIN
 A[I,J] := 99;
END;
```

In the next example, the Pascal routine passes the character string "0123456789" to the Fortran subroutine **S_**, which prints it out and then returns to the calling program.

Pascal

```
TYPE STRING = ARRAY[1..10] OF CHAR;
PROCEDURE S_( VAR A: STRING; I: INTEGER); EXTERNAL;
/* Note the underbar */
PROGRAM TEST;
VAR
 R: STRING;
BEGIN
 R:= "0123456789";
 S_(R,10);
END.
```

Fortran

```
   SUBROUTING S(C)
   CHARACTER*10 C
   WRITE (6,10) C
10 FORMAT (6,10) C
   RETURN
   END
```

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- Fortran common blocks must be declared by *common* statements; Pascal can use any global variable. Note that the common block name in Pascal (**sam_**) must end with an underscore.
- Data types in the Fortran and Pascal programs must match unless you want implicit equivalencing. If so, adhere to the alignment restrictions for the data types described in Chapter 2, "Storage Mapping."
- If the same common block is of unequal length, the largest size is used to allocate space.
- Unnamed common blocks are given the name **_BLNK_**, where **_** is the underscore character.

Example

The following examples show Fortran and Pascal routines that access common blocks of data.

Pascal

```
VAR
A_: RECORD
    I : INTEGER;
    R : REAL;
END;
PROCEDURE SAM_;
 EXTERNAL;
PROGRAM S;
BEGIN
A_.I := 4;
A_.R := 5.3;
SAM_;
END.
```

Fortran

```
   SUBROUTINE SAM()
   COMMON /A/I,R
   WRITE (6,10) i,r
10 FORMAT (1x,I5,F5.2)
   RETURN
   END
```

The Fortran routine prints out 4 and 5.30.