Chapter 3

Fortran Program Interfaces

Sometimes it is necessary to create a program that combines modules written in Fortran and another language. For example,

- In a Fortran program, you need access to a facility that is only available as a C function, such as a member of a graphics library.
- In a program in another language, you need access to a computation that has been implemented as a Fortran subprogram, for example one of the many well-tested, efficient routines in the BLAS library.

Tip: Fortran subroutines and functions that give access to the IRIX system functions and other IRIX facilities already exist, and are documented in Chapter 4 of this manual.

This chapter focuses on the interface between Fortran and the most common other language, C. However other language can be called, for example C++.

Note: You should be aware that all compilers for a given version of IRIX use identical standard conventions for passing parameters in generated code. These conventions are documented at the machine instruction level in the *MIPSpro Assembly Language Programmer's Guide*, which also details the differences in the conventions used in different releases.

How Fortran Treats Subprogram Names

The Fortran compiler normally changes the names of subprograms and named common blocks while it translates the source file. When these names appear in the object file for reference by other modules, they are normally changed in two ways:

- converted to all lowercase letters
- extended with a final underscore (_) character

Normally the following declarations

```
SUBROUTINE MATRIX
function MixedCase()
COMMON /CBLK/a,b,c
```

produce the identifiers **matrix_**, **mixedcase_**, and **cblk_** (all lowercase with appended underscore) in the generated object file.

Note: Fortran intrinsic functions are not named according to these rules. The external names of intrinsic functions as defined in the Fortran library are not directly related to the intrinsic function names as they are written in a program. The use of intrinsic function names is discussed in the *MIPSpro Fortran 77 Language Reference Manual*.

Working with Mixed-Case Names

There is no way by which you can make the Fortran compiler generate an external name containing uppercase letters. If you are porting a program that depends on the ability to call such a name, you will have to write a C function that takes the same arguments but which has a name composed of lowercase letters only. This C function can then call the function whose name contains mixed-case letters.

Note: Previous versions of the Fortran 77 compiler for 32-bit systems supported the -U compiler option, telling the compiler to not force all uppercase input to lowercase. As a result, uppercase letters could be preserved in external names in the object file. As now implemented, this option does not affect the case of external names in the object file.

Preventing a Suffix Underscore with $

You can prevent the compiler from appending an underscore to a name by writing the name with a terminal currency symbol ($). The '$' is not reproduced in the object file. It is dropped, but it prevents the compiler from appending an underscore. The declaration

```
EXTERNAL NOUNDER$
```

produces the name **nounder** (lowercase, but no trailing underscore) in the object file.

Note: This meaning of '$' in names applies only to subprogram names. If you end the name of a COMMON block with '$,' the name in the object file includes the '$' and ends with an underscore regardless.

Naming Fortran Subprograms from C

In order to call a Fortran subprogram from a C module you must spell the name the way the Fortran compiler spells it—normally, using all lowercase letters and a trailing underscore. A Fortran subprogram declared as follows:

```
SUBROUTINE HYPOT()
```

would typically be declared in this C function (lowercase with trailing underscore):

```
extern int hypot_()
```

You must know if a subprogram is declared with a trailing '$' to suppress the underscore.

Naming C Functions from Fortran

The C compiler does not modify the names of C functions. C functions can have uppercase or mixed-case names, and they have terminal underscores only when the programmer writes them that way.

In order to call a C function from a Fortran program you must ensure that the Fortran compiler spells the name correctly. When you control the name of the C function, the simplest solution is to give it a name that consists of lowercase letters with a terminal underscore. For example, the following C function:

```
int fromfort_() {...}
```

could be declared in a Fortran program as follows:

```
EXTERNAL FROMFORT
```

When you do not control the name of a C function, you must cause the Fortran compiler to generate the correct name in the object file. Write the C function's name using a terminal '$' character to suppress the terminal underscore. (You cannot cause the compiler to generate an external name with uppercase letters in it.)

Testing Name Spelling Using *nm*

You can verify the spelling of names in an object file using the *nm* command (or with the elfdump command with the *-t* or *-Dt* options). To see the subroutine and common names generated by the compiler, apply *nm* to the generated .o (object) or executable file.

Correspondence of Fortran and C Data Types

When you exchange data values between Fortran and C, either as parameters, as function results, or as elements of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data value.

Corresponding Scalar Types

The correspondence between Fortran and C scalar data types is shown in Table 3-1. This table assumes the default precisions. Use of compiler options such as *-i2* or *-r8* affects the meaning of the words LOGICAL, INTEGER, and REAL.

Table 3-1 Corresponding Fortran and C Data Types

Fortran Data Type	Corresponding C type
BYTE, INTEGER*1, LOGICAL*1	signed char
CHARACTER*1	unsigned char
INTEGER*2, LOGICAL*2	short
INTEGER[a], INTEGER*4, LOGICAL[a], LOGICAL*4	int or long
INTEGER*8, LOGICAL*8	long long
REAL[a], REAL*4	float
DOUBLE PRECISION, REAL*8	double
REAL*16	long double
COMPLEX[a], COMPLEX*8	typedef struct{float real, imag; } cpx8;
DOUBLE COMPLEX, COMPLEX*16	typedef struct{ double real, imag; } cpx16;
COMPLEX*32	typedef struct{long double real, imag;} cpx32;
CHARACTER**n* (*n*>1)	typedef char fstr_*n*[*n*];

a. Assuming default precision

The rules governing alignment of variables within common blocks are covered under "Alignment, Size, and Value Ranges" on page 21.

Corresponding Character Types

The Fortran CHARACTER*1 data type corresponds to the C type unsigned char. However, the two languages differ in the treatment of strings of characters.

A Fortran CHARACTER*n (n>1) variable contains exactly n characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach n characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran string. (The programmer can create a null byte using the Hollerith constant '\0' but this is not normally done.)

Since there is no terminal null byte, most of the string library functions familiar to C programmers (**strcpy()**, **strcat()**, **strcmp()**, and so on) cannot be used with Fortran string values. The **strncpy()**, **strncmp()**, **bcopy()**, and **bcmp()** functions can be used because they depend on a count rather than a delimiter.

Corresponding Array Elements

Fortran and C use different arrangements for the elements of an array in memory. Fortran uses column-major order (when iterating sequentially through memory, the leftmost subscript varies fastest), whereas C uses row-major order (the rightmost subscript varies fastest to generate sequential storage locations). In addition, Fortran array indices are normally origin-1, while C indices are origin-0.

To use a Fortran array in C,

- Reverse the order of dimension limits when declaring the array
- Reverse the sequence of subscript variables in a subscript expression
- Adjust the subscripts to origin-0 (usually, decrement by 1)

The correspondence between Fortran and C subscript values is depicted in Figure 3-1. You derive the C subscripts for a given element by decrementing the Fortran subscripts and using them in reverse order; for example, Fortran (99,9) corresponds to C [8][98].

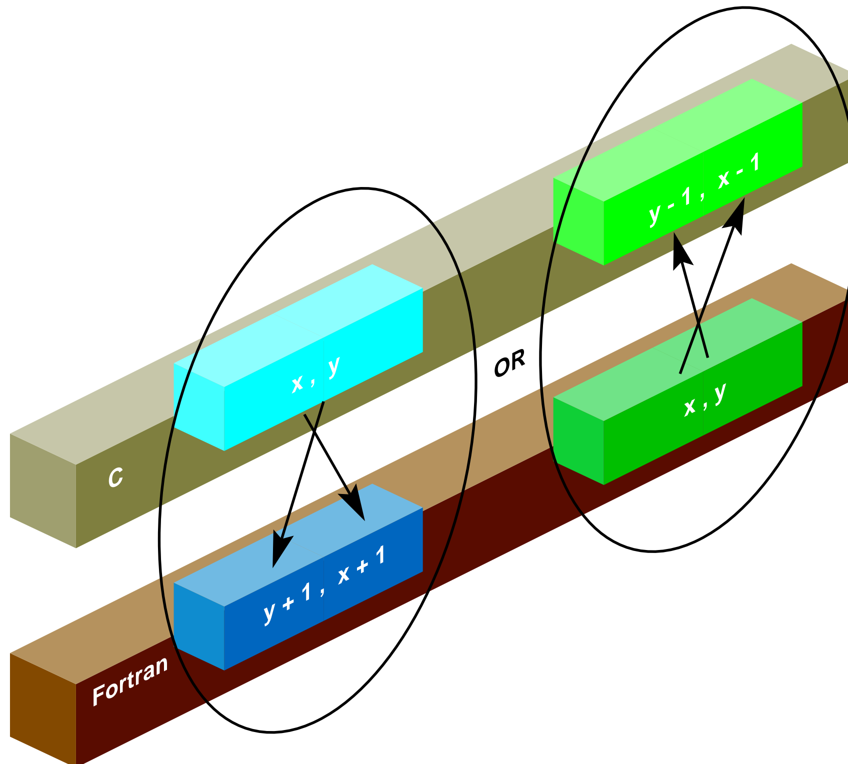

Figure 3-1 Correspondence Between Fortran and C Array Subscripts

For a coding example, see "Using Fortran Arrays in C Code" on page 41.

Note: A Fortran array can be declared with some other lower bound than the default of 1. If the Fortran subscript is origin-0, no adjustment is needed. If the Fortran lower bound is greater than 1, the C subscript is adjusted by that amount.

How Fortran Passes Subprogram Parameters

The Fortran compiler generates code to pass parameters according to simple, uniform rules; and it generates subprogram code that expects parameters to be passed according to these rules. When calling non-Fortran functions, you must know how parameters will be passed; and when calling Fortran subprograms from other languages you must cause the other language to pass parameters correctly.

Normal Treatment of Parameters

Every parameter passed to a subprogram, regardless of its data type, is passed as the address of the actual parameter value in memory. This rule is extended for two cases:

- The length of each CHARACTER**n* parameter (when *n*>1) is passed as an additional, INTEGER value, following the explicit parameters.
- When a function returns type CHARACTER**n* parameter (*n*>1), the address of the space to receive the result is passed as the fi rst parameter to the function and the length of the result space is passed as the second parameter, preceding all explicit parameters.

Example 3-1 Example Subroutine Call

```
COMPLEX*8 cp8
CHARACTER*16 creal, cimag
CALL CPXASC(creal,cimag,cp8)
```

Code generated from the CALL in Example 3-1 prepares these 5 argument values:

1. The address of *creal*
2. The address of *cimag*
3. The address of *cp8*
4. The length of *creal*, an integer value of 16
5. The length of *cimag*, an integer value of 16

Example 3-2 Example Function Call

```
CHARACTER*8 symbl,picksym
CHARACTER*100 sentence
INTEGER nsym
symbl = picksym(sentence,nsym)
```

Code generated from the function call in Example 3-2 prepares these 5 argument values:

1. The address of variable *symbl*, the function result space
2. The length of *symbl*, an integer value of 8
3. The address of *sentence*, the first explicit parameter
4. The address of *nsym*, the second explicit parameter
5. The length of *sentence*, an integer value of 100

You can force changes in these conventions using %VAL and %LOC; this is covered under "Calls to C Using LOC%, REF% and VAL%" on page 42.

Calling Fortran from C

There are two types of callable Fortran subprograms: subroutines and functions (these units are documented in the *MIPSpro Fortran 77 Language Reference Manual*). In C terminology, both types of subprogram are external functions. The difference is the use of the function return value from each.

Calling Fortran Subroutines from C

From the standpoint of a C module, a Fortran subroutine is an external function returning int. The integer return value is normally ignored by a C caller (its meaning is discussed in "Alternate Subroutine Returns" on page 36).

The following two examples show a simple Fortran subroutine and a sketch of a call to it.

Example 3-3 Example Fortran Subroutine with COMPLEX Parameters

```
SUBROUTINE ADDC32(Z,A,B,N)
COMPLEX*32 Z(1),A(1),B(1)
INTEGER N,I
DO 10 I = 1,N
    Z(I) = A(I) + B(I)
10 CONTINUE
RETURN
END
```

Example 3-4 C Declaration and Call with COMPLEX Parameters

```
typedef struct{long double real, imag;} cpx32;
extern int
  addc32_(cpx32*pz,cpx32*pa,cpx32*pb,int*pn);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
...
    int n = MAXARRAY;
    (void)addc32_(&z, &a, &b, &n);
```

The Fortran subroutine in Example 3-3 is named in Example 3-4 using lowercase letters and a terminal underscore. It is declared as returning an integer. For clarity, the actual call is cast to (void) to show that the return value is intentionally ignored.

The trivial subroutine in the following example takes adjustable-length character parameters.

Example 3-5 Example Fortran Subroutine with String Parameters

```
SUBROUTINE PRT(BEF,VAL,AFT)
CHARACTER*(*)BEF,AFT
REAL VAL
PRINT *,BEF,VAL,AFT
RETURN
END
```

Example 3-6 C Program that Passes String Parameters

```
typedef char fstr_16[16];
extern int
   prt_(fstr_16*pbef, float*pval, fstr_16*paft,
                int lbef, int laft);
main()
{
    float val = 2.1828e0;
    fstr_16 bef,aft;
    strncpy(bef,"Before..........",sizeof(bef));
    strncpy(aft,"..........After",sizeof(aft));
    (void)prt_(bef,&val,aft,sizeof(bef),sizeof(aft));
}
```

The C program in Example 3-6 prepares CHARACTER*16 values and passes them to the subroutine in Example 3-5. Observe that the subroutine call requires 5 parameters, including the lengths of the two string parameters. In Example 3-6, the string length parameters are generated using **sizeof()**, derived from the typedef *fstr_16*.

Example 3-7 C Program that Passes Different String Lengths

```
extern int
prt_(char*pbef, float*pval, char*paft, int lbef, int laft);
main()
{
        float val = 2.1828e0;
        char *bef = "Start:";
        char *aft = ":End";
        (void)prt_(bef,&val,aft,strlen(bef),strlen(aft));
}
```

When the Fortran code does not require a specific length of string, the C code that calls it can pass an ordinary C character vector, as shown in Example 3-7. In Example 3-7, the string length parameter length values are calculated dynamically using **strlen()**.

Alternate Subroutine Returns

In Fortran, a subroutine can be defined with one or more asterisks (*) in the position of dummy parameters. When such a subroutine is called, the places of these parameters in the CALL statement are supposed to be filled with statement numbers or statement labels. The subroutine returns an integer which selects among the statement numbers, so that the subroutine call acts as both a call and a computed go-to (for more details, see the discussions of the CALL and RETURN statements in the *MIPSpro Fortran 77 Language Reference Manual*).

Fortran does not generate code to pass statement numbers or labels to a subroutine. No actual parameters are passed to correspond to dummy parameters given as asterisks. When you code a C prototype for such a subroutine, simply ignore these parameter positions. A CALL statement such as

```
CALL NRET (*1,*2,*3)
```

is treated exactly as if it were the computed GOTO written as

```
GOTO (1,2,3), NRET()
```

The value returned by a Fortran subroutine is the value specifi ed on the RETURN statement, and will vary between 0 and the number of asterisk dummy parameters in the subroutine defi nition.

Calling Fortran Functions from C

A Fortran function returns a scalar value as its explicit result. This corresponds exactly to the C concept of a function with an explicit return value. When the Fortran function returns any type shown in Table 3-1 other than CHARACTER**n* (*n*>1), you can call the function from C and handle its return value exactly as if it were a C function returning that data type.

Example 3-8 Fortran Function Returning COMPLEX*16

```
COMPLEX*16 FUNCTION FSUB16(INP)
COMPLEX*16 INP
FSUB16 = INP
END
```

The trivial function shown in Example 3-8 accepts and returns COMPLEX*16 values. Although a COMPLEX value is declared as a structure in C, it can be used as the return type of a function.

Example 3-9 C Program that Receives COMPLEX Return Value

```
typedef  struct{ double real, imag; } cpx16;
extern cpx16 fsub16_( cpx16 * inp );
main()
{
    cpx16 inp = { -3.333, -5.555 };
    cpx16 oup = { 0.0, 0.0 };
    printf("testing fsub16...");
    oup = fsub16_( &inp );
    if ( inp.real == oup.real && inp.imag == oup.imag )
        printf("Ok\n");
    else
        printf("Nope\n");
}
```

The C program in Example 3-9 shows how the function in Example 3-8 is declared and called. Observe that the parameters to a function, like the parameters to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Note: In IRIX 5.3 and earlier, you can *not* call a Fortran function that returns COMPLEX (although you can call one that returns any other arithmetic type). The register conventions used by compilers prior to IRIX 6.0 do not permit returning a structure value from a Fortran function to a C caller.

Example 3-10 Fortran Function Returning CHARACTER*16

```
CHARACTER*16 FUNCTION FS16(J,K,S)
CHARACTER*16 S
INTEGER J,K
FS16 = S(J:K)
RETURN
END
```

The function in Example 3-10 has a CHARACTER*16 return value. When the Fortran function returns a CHARACTER*n (n>1) value, the returned value is not the explicit result of the function. Instead, you must pass the address and length of the result area as the first two parameters of the function.

Example 3-11 C Program that Receives CHARACTER*16 Return

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *pz,int lz,int *pj,int *pk,fstr_16*ps,int ls);
main()
{
    char work[64];
    fstr_16 inp,oup;
    int j=7;
    int k=11;
    strncpy(inp,"0123456789abcdef",sizeof(inp));
    fs16_ ( oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
    strncpy(work,oup,sizeof(oup));
    work[sizeof(oup)] = '\0';
    printf("FS16 returns <%s>\n",work);
}
```

The C program in Example 3-11 calls the function in Example 3-10. The address and length of the function result are the first two parameters of the function. (Since type *fstr_16* is an array, its name, *oup*, evaluates to the address of its first element.) The next three parameters are the addresses of the three named parameters; and the final parameter is the length of the string parameter.

Calling C from Fortran

In general, you can call units of C code from Fortran as if they were written in Fortran, provided that the C modules follow the Fortran conventions for passing parameters (see "How Fortran Passes Subprogram Parameters" on page 33). When the C program expects parameters passed using other conventions, you can either write special forms of CALL, or you can build a "wrapper" for the C functions using the *mkf2c* command.

Normal Calls to C Functions

The C function in this section is written to use the Fortran conventions for its name (lowercase with final underscore) and for parameter passing.

Example 3-12 C Function Written to be Called from Fortran

```
/*
|| C functions to export the facilities of strtoll()
|| to Fortran 77 programs.  Effective Fortran declaration:
||
|| INTEGER*8 FUNCTION ISCAN(S,J)
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the nonspace character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
   int  scanPos, scanLen;
   long long ret = 0;
   char wrk[1024];
   char *endpt;
   /* when J>LEN(S), do nothing, return 0 */
```

```
   if (ls >= *pj)
   {
      /* convert J to origin-0, permit J=0 */
      scanPos = (0 < *pj)? *pj-1 : 0 ;

      /* calculate effective length of S(J:) */
      scanLen = ls - scanPos;

      /* copy S(J:) and append a null for strtoll() */
      strncpy(wrk,(ps+scanPos),scanLen);
      wrk[scanLen] = '\0';

      /* scan for the integer */
      ret = strtoll(wrk, &endpt, 0);

      /*
      || Advance over any whitespace following the number.
      || Trailing spaces are common at the end of Fortran
      || fixed-length char vars.
      */
      while(isspace(*endpt)) { ++endpt; }
      *pj = (endpt - wrk)+scanPos+1;
   }
   return ret;
}
```

The following program in demonstrates a call to the function in Example 3-12.

```
EXTERNAL ISCAN
INTEGER*8 ISCAN
INTEGER*8 RET
INTEGER J,K
CHARACTER*50 INP
INP = '1  -99   3141592  0xfff  033 '
J = 0
DO 10 WHILE (J .LT. LEN(INP))
    K = J
    RET = ISCAN(INP,J)
    PRINT *, K,': ',RET,' -->',J
10  CONTINUE
END
```

Using Fortran COMMON in C Code

A C function can refer to the contents of a COMMON block defined in a Fortran program. The name of the block as given in the COMMON statement is altered as described in "How Fortran Treats Subprogram Names" on page 27 (that is, forced to lowercase and extended with an underscore). The name of the "blank common" is _BLNK__ (one leading, two final, underscores).

In order to refer to the contents of a common block, take these steps:

- Declare a structure whose fields have the appropriate data types to match the successive elements of the Fortran common block. (See Table 3-1 for corresponding data types.)
- Declare the common block name as an external structure of that type.

An example is shown below.

Example 3-13 Common Block Usage in Fortran and C

```
      INTEGER STKTOP,STKLEN,STACK(100)
      COMMON /WITHC/STKTOP,STKLEN,STACK

struct fstack {
    int stktop, stklen;
    int stack[100];
}
extern fstack withc_;
int peektop_()
{
    if (withc_.stktop) /* stack not empty */
        return withc_.stack[withc_.stktop-1];
    else...
}
```

Using Fortran Arrays in C Code

As described under "Corresponding Array Elements" on page 31, a C program must take special steps to access arrays created in Fortran.

Example 3-14 Fortran Program Sharing an Array in Common with C

```
INTEGER IMAT(10,100),R,C
COMMON /WITHC/IMAT
R = 74
C = 6
CALL CSUB(C,R,746)
PRINT *,IMAT(6,74)
END
```

The Fortran fragment in Example 3-14 prepares a matrix in a common block, then calls a C subroutine to modify the array.

Example 3-15 C Subroutine to Modify a Common Array

```
extern struct { int imat[100][10]; } withc_;
int csub_(int *pc, int *pr, int *pval)
{
    withc_.imat[*pr-1][*pc-1] = *pval;
    return 0; /* all Fortran subrtns return int */
}
```

The C function in Example 3-15 stores its third argument in the common array using the subscripts passed in the fi rst two agruments. In the C function, the order of the dimensions of the array are reversed. The subscript values are reversed to match, and decremented by 1 to match the C assumption of 0-origin indexing.

Calls to C Using LOC%, REF% and VAL%

Using the special intrinsic functions %VAL, %REF, and %LOC you can pass parameters in ways other than the standard Fortran conventions described under '" How Fortran Passes Subprogram Parameters"on page 33. These intrinsic functions are documented in the *MIPSpro Fortran 77 Language Reference Manual*.

Using %VAL

%VAL is used in parameter lists to cause parameters to be passed by value rather than by reference. Examine the following function prototype (from the random(3b) reference page).

```
char *initstate(unsigned int seed, char *state, int n);
```

This function takes an integer value as its fi rst parameterFortran would normally pass the address of an integer value, but %VAL can be used to make it pass the integer itself. Example 3-16 demonstrates a call to function **initstate()** and the other functions of the **random()** group.

Example 3-16 Fortran Function Calls Using %VAL

```
C declare the external functions in random(3b)
C random() returns i*4, the others return char*
      EXTERNAL RANDOM$, INITSTATE$, SETSTATE$
      INTEGER*4 RANDOM$
      INTEGER*8 INITSTATE$,SETSTATE$
C We use "states" of 128 bytes, see random(3b)
C Note: An undocumented assumption of random() is that
C a "state" is dword-aligned!  Hence, use a common.
      CHARACTER*128 STATE1, STATE2
      COMMON /RANSTATES/STATE1,STATE2
C working storage for state pointers
      INTEGER*8 PSTATE0, PSTATE1, PSTATE2
C initialize two states to the same value
      PSTATE0 = INITSTATE$(%VAL(8191),STATE1)
      PSTATE1 = INITSTATE$(%VAL(8191),STATE2)
      PSTATE2 = SETSTATE$(%VAL(PSTATE1))
C pull 8 numbers from state 1, print
      DO 10 I=1,8
         PRINT *,RANDOM$()
10    CONTINUE
C set the other state, pull 8 numbers & print
      PSTATE1 = SETSTATE$(%VAL(PSTATE2))
      DO 20 I=1,8
         PRINT *,RANDOM$()
20    CONTINUE
      END
```

The use of %VAL(8191) or %VAL(PSTATE1) causes that value to be passed, rather than an address of that value.

Using %REF

%REF is used in parameter lists to cause parameters to be passed by reference, that is, to pass the address of a value rather than the value itself.

Passing parameters by reference is the normal behavior of Silicon Graphics Fortran 77 compilers, so no effective difference exists between writing %REF(*parm*) and writing *parm* alone in a parameter list for non-character parameters. For character parameters, using %REF(*parm*) causes the character string length not to be added to the end of the parameter list as in the normal case. Thus, using the %REF(*parm*) guarantees that only the address of the parameter is parsed and nothing else.

When calling a C function that expects the address of a value rather than the value itself, you can write %REF(*parm*). Examine this C prototype (see the gmatch(3G) reference page).

```
int gmatch (const char *str, const char *pattern);
```

This function **gmatch()** could be declared and called from Fortran.

Example 3-17 Fortran Call to *gmatch()* Using %REF

```
LOGICAL GMATCH$
CHARACTER*8 FNAME,FPATTERN
FNAME = 'foo.f\0'
FPATTERN = '*.f\0'
IF ( GMATCH$(%REF(FNAME),%REF(FPATTERN)) )...
```

The use of %REF() in Example 3-17 simply documents the fact that *gmatch()* expects addresses of character strings.

Using %LOC

%LOC returns the address of its argument. It can be used in any expression (not only within parameter lists), and is often used to set POINTER variables.

Making C Wrappers with *mkf2c*

The program *mkf2c* provides an alternate interface for C routines called by Fortran. (Some details of *mkf2c* are covered in the mkf2c(1) reference page.)

The *mkf2c* program reads a file of C function prototype declarations and generates an assembly language module. This module contains one callable entry point for each C function. The entry point, or "wrapper," accepts parameters in the Fortran calling convention, and passes the same values to the C function using the C conventions.

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

```
simplefunc (int a, double df)
{ /* function body ignored */ }
```

For this function, *mkf2c* (with no options) generates a wrapper function named **simplefunc_** (with an underscore appended). The wrapper function expects two parameters, an integer and a REAL*8, passed according to Fortran conventions; that is, by reference. The code of the wrapper loads the values of the parameters into registers using C conventions for passing parameters by value, and calls **simplefunc()**.

Parameter Assumptions by *mkf2c*

Since *mkf2c* processes only the C source, not the Fortran source, it treats the Fortran parameters based on the data types specified in the C function header. These treatments are summarized in Table 3-2.

Note: Through compiler release 6.0.2, *mkf2c* does not recognize the C data types "long long" and "long double" (INTEGER*8 and REAL*16). It treats arguments of this type as "long" and "double" respectively.

Table 3-2 How *mkf2c* treats Function Arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
unsigned char	Load CHARACTER*1 from memory to register, no sign extension
char	Load CHARACTER*1 from memory to register; sign extension only when *-signed* is specified
unsigned short, unsigned int	Load INTEGER*2 or INTEGER*4 from memory to register, no sign extension
short	Load INTEGER*2 from memory to register with sign extension
int, long	Load INTEGER*4 from memory to register with sign extension
long long	(Not supported through 6.0.2)
float	Load REAL*4 from memory to register, extending to double unless *-f* is specified
double	Load REAL*8 from memory to register
long double	(Not supported through 6.0.2)

Table 3-2 (continued) How *mkf2c* treats Function Arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
char *name*[], *name*[*n*]	Pass address of CHARACTER***n* and pass length as integer parameter as Fortran does
char *	Copy CHARACTER***n* value into allocated space, append null byte, pass address of copy

Character String Treatment by *mkf2c*

In Table 3-2, notice the different treatments for an argument declared as a character array and one declared as a character address (even though these two declarations are semantically the same in C).

When the C function expects a character address, *mkf2c* generates the code to dynamically allocate memory and to copy the Fortran character value, for its specified length, to the allocated memory. This creates a null-terminated string. In this case,

- The address passed to C points to the allocated memory
- The length of the value is not passed as an implicit argument
- There is a terminating null byte in the value
- Changes in the string are *not* reflected back to Fortran

A character array specified in the C function is processed by *mkf2c* just like a Fortran CHARACTER***n* value. In this case,

- The address prepared by Fortran is passed to the C function
- The length of the value is passed as an implicit argument (see "Normal Treatment of Parameters" on page 33)
- The character array contains no terminating null byte (unless the Fortran programmer supplies one)
- Changes in the array by the C function will be visible to Fortran

Since the C function cannot declare the extra string-length parameter (if it declared the parameter, *mkf2c* would process it as an explicit argument) the C programmer has a choice of ways to access the string length. When the Fortran program always passes character values of the same size, the length parameter can simply be ignored. If its value is needed, the **varargs** macro can be used to retrieve it.

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

```
void func1 (char carr1[],int i, char *str, char carr2[]);
```

mkf2c passes a total of six parameters to C. The fifth parameter is the length of the Fortran value corresponding to *carr1*. The sixth is the length of *carr2.* The C function can use the **varargs** macros to retrieve these hidden parameters. *mkf2c* ignores the *varargs* macro *va_alist* appearing at the end of the parameter name list.

When func1 is changed to use varargs, the C source file is as follows.

Example 3-18 C Function Using *varargs*

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

The C routine would retrieve the lengths of *carr1* and *carr2,* placing them in the local variables *carr1_len* and *carr2_len* using code like the following fragment.

Example 3-19 C Code to Retrieve Hidden Parameters

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

Restrictions of *mkf2c*

When it does not recognize the data type specified in the C function, *mkf2c* issues a warning message and generates code to simply pass the pointer passed by Fortran. It does this in the following cases:

- Any nonstandard data type name, for example a data type that might be declared using typedef or a data type defined as a macro
- Any structure argument
- Any argument with multiple indirection (two or more asterisks, for example *char***)

Since *mkf2c* does not support structure-valued arguments, it does not support passing COMPLEX**n* values.

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 does not include constructs such as typedefs, external function declarations, or C preprocessor directives.

To ensure that only the 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 the 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 fi le for*mkf2c*.

Example 3-20 Source File for Use with *extcentry*

```
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 */
```

Example 3-20 illustrates the use of *extcentry*. It shows the C fi le*foo.c* containing the function **foo**, which is to be made Fortran callable.

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 fi le*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*.

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 controls 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*.