SUN/209.11

Starlink Project
Starlink User Note 209.11

P.M. Allan
A.J. Chipperfield
R.F. Warren-Smith
P.W. Draper
D.S. Berry

12 May 2011

CNF and F77 Mixed Language Programming – FORTRAN and C Version 4.3 Programmer's Manual

Abstract

The CNF package comprises two sets of software which ease the task of writing portable programs in a mixture of FORTRAN and C. F77 is a set of C macros for handling the FORTRAN/C subroutine linkage in a portable way, and CNF is a set of functions to handle the difference between FORTRAN and C character strings, logical values and pointers to dynamically allocated memory.

Contents

1 How to read this document

This document tells a programmer how to mix program segments written in FORTRAN and C *in a portable way*. It provides information on several levels from a quick "how to get started" cookbook, down to machine-specific details. The cookbook will tell you how to write programs, but without much of the background information of what is really going on. After you have tried a few programs, you will probably want to read the rest of the document.

Before you embark upon mixed language programming, it may be worth reading the Rationale for mixed language programming in Appendix B which discusses the problems and offers some alternatives.

The current system is supported for Sun systems running Solaris, DEC Alphas running OSF/1, and PC's running Linux but in the past has run successfully on SunOS, Ultrix and VAX/VMS. Reference is made to the VAX/VMS system in this document as it is in many respects very different from the Unix systems and so provides a useful comparison. You should consult the VMS Starlink documentation set about the VAX/VMS version however, as not all the facilities described here are available in it, *even if a VAX/VMS example is given*.

Full descriptions of the C macros and functions involved are

provided in appendices E and G. The macro names in the text will often include the legend `type` to indicate a generic macro name. In this case, `type` may normally be one of `INTEGER`, `REAL`, `DOUBLE`, `LOGICAL` or `CHARACTER`. Types `BYTE` and `WORD` and their unsigned versions `UBYTE` and `UWORD` are also available but do not correspond to standard FORTRAN types so should be avoided. `type` may also be `POINTER` – again this is not a standard FORTRAN type but it is more commonly used in Starlink software (see Section 7).

For consistency with other Starlink libraries the CNF function names were changed (at Version 4.0) from the form `cnf_name` to the form `cnfName`. The old names are still permitted via macros defined in the `f77.h` header file.

There is also a section (13) on how to compile and link the programs.

2 Cookbook

This section introduces mixed language programming. It skips over many of the details and concentrates on how to get programs going. For a fuller explanation of mixed language programming, you should read the rest of this document.

2.1 Calling C from FORTRAN

Why would you want to call a C function from a FORTRAN program? Typically this will be to do something in the C function that cannot be done from FORTRAN, at least not in the way that you would like. On account of this, realistic examples of calling C from FORTRAN can be rather involved. After all, you can do most simple things from FORTRAN itself. So as not to obscure how to go about writing mixed language programs with complex C functions, the examples in this section concentrate on what to do when mixing C and FORTRAN rather than on providing realistic examples of this.

Here is an example of a FORTRAN program that calls a C function which sets various arguments.

Example 1 – Calling C from FORTRAN.

FORTRAN program:

```
      PROGRAM COOK1
      INTEGER I,J
      REAL A,B
      CHARACTER*(80) LINE
      LOGICAL X

      I = 1
      A = 5.0
      X = .FALSE.
      LINE = ' '
      CALL SILLY1( A, B, I, J, LINE, LEN(LINE), X )
      PRINT *, LINE

      END
```

C function:

```
#include "f77.h"

F77_SUBROUTINE(silly1)( REAL(a), REAL(b), INTEGER(i), INTEGER(j),
  CHARACTER(line), INTEGER(line_l), LOGICAL(x) TRAIL(line) )
{
  GENPTR_REAL(a)
  GENPTR_REAL(b)
  GENPTR_INTEGER(i)
  GENPTR_INTEGER(j)
  GENPTR_CHARACTER(line)
  GENPTR_INTEGER(line_l)
  GENPTR_LOGICAL(x)

  char str[] = "This is a string";

  if( F77_ISTRUE(*x) )
  {
     *b = *a;
     *j = *i;
  }
  else
  {
     cnfExprt( str, line, *line_l );
  }
}
```

This is a rather silly example, but it does illustrate all of the important points of calling C from FORTRAN. The FORTRAN program is completely standard. The name of the C function is declared using a macro `F77_SUBROUTINE`. Do not leave any spaces around the name of the routine as this can cause problems on some systems. The dummy arguments of the function are declared using macros named after the FORTRAN type of the actual argument. The only odd thing is the macro called `TRAIL`. Each argument of type `CHARACTER` should have a corresponding `TRAIL` added to the end of the argument list. N.B. `TRAIL` macros must not have a comma in front of them. All C functions that are to be called from FORTRAN should be declared in a similar manner.

There then follows a set of `GENPTR_`*type* macros; one for each argument of the function. `TRAIL` arguments are not counted as being true arguments and so there are no `GENPTR` statements for them. Note that there are no semicolons at the end of these lines.

The only other macro used is `F77_ISTRUE`. This should be used whenever an argument is treated as a logical value, and takes into account the different ways that FORTRAN and C may interpret bit patterns as logical values.

Note that all explicit function arguments are pointer arguments. This is necessary if their value is to be modified in the function. The consequence of this is that scalar arguments must be referred to by `*`*arg* within the function.

FORTRAN and C store character strings in different ways. FORTRAN stores them as fixed-length, blank-filled strings while C stores them as variable-length, null-terminated strings. If a C function needs to work with character strings that have been passed from a calling FORTRAN routine, then the FORTRAN string must be copied into an equivalent local copy. Similarly, a C function may need to return a string to the calling FORTRAN routine. This is a very common occurrence, so some “CNF functions” are provided to do this. Essentially they are just C functions which copy a FORTRAN string to a C string and vice versa. (They are more fully described in Section 6.1.)

In the above example, the function `cnfExprt` copies the C string `str` into the FORTRAN string `line`. Function `cnfImprt` performs the converse operation in “Calling FORTRAN from C” example in Section 2.2.

2.2 Calling FORTRAN from C

Why would you want to call a FORTRAN subprogram from a C routine? Typically this would be because you want to use a precompiled library of routines that were written in FORTRAN. The NAG library is a prime example. This can be bought in a C callable version, but this is not available on Starlink machines.

To see how to call FORTRAN from C, let us consider the above example, but now with the roles of FORTRAN and C exchanged.

Example 2 – Calling FORTRAN from C.

C program:

```
#include "f77.h"

#define FLINE_LENGTH 80

extern F77_SUBROUTINE(silly2)( REAL(a), REAL(b), INTEGER(i), INTEGER(j),
  CHARACTER(line), INTEGER(line_l), LOGICAL(x) TRAIL(line) );

main()
{
  DECLARE_INTEGER(i);
  DECLARE_INTEGER(j);
  DECLARE_INTEGER(fline_l);
  DECLARE_REAL(a);
  DECLARE_REAL(b);
  DECLARE_LOGICAL(x);
  DECLARE_CHARACTER(fline,FLINE_LENGTH);

  char line[FLINE_LENGTH+1];

  fline_l = FLINE_LENGTH;

  i = 1;
  a = 5.0;
  x = F77_FALSE;

  F77_CALL(silly2)( REAL_ARG(&a), REAL_ARG(&b), INTEGER_ARG(&i),
    INTEGER_ARG(&j), CHARACTER_ARG(fline), INTEGER_ARG(&fline_l),
    LOGICAL_ARG(&x) TRAIL_ARG(fline) );

  cnfImprt( fline, FLINE_LENGTH, line );

  printf( "%s\n", line );
}
```

FORTRAN function:

```
      SUBROUTINE SILLY2( A, B, I, J, LINE, LINE_L, X )
      REAL A, B
      INTEGER I, J
      CHARACTER * ( * ) LINE
      INTEGER LINE_L
      LOGICAL X

      IF( X ) THEN
        B = A
        J = I
      ELSE
         LINE = 'This is a string'
      END IF

      END
```

In the above C main program, the variable `fline_l` is declared and set equal to the constant `FLINE_LENGTH`. At first sight this is unnecessary. However, this is not the case, as we need to pass the value of `FLINE_LENGTH` to the subroutine and it is not possible to pass constants to FORTRAN subroutines. Only variables can be passed.

2.3 Building the Program

The final step is compiling and linking the program.

Suppose, on Unix, the main FORTRAN program is in the file `cook1.f` and the C function is in the file `silly1.c`, then the commands might be:

```
% cc -c -I/star/include silly1.c
% f77 cook1.f silly1.o -L/star/lib ‘cnf_link‘
```

Note that the compiling and linking commands are somewhat machine-specific – compiling the FORTRAN routine first and then trying to link the routine using the `cc` command generally does not work. More details are given in Compiling and Linking (see Section 13).

Armed with the above examples, you should be in a position to start experimenting with mixed language programming. For further information, read on.

3 Representation of Data

Different languages have differing fundamental data types on which they can operate. FORTRAN has the types `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL` and `CHARACTER`. The only aggregate data type that it supports is the array, although a character variable can store many characters. C supports the fundamental types `int`, `float`, `double`, `char`, and `void`. It also allows `int` to be modified by the type specifiers `short` or `long`, `signed` or `unsigned`, `char` to be modified by `signed` or `unsigned` and `double` to be modified by `long`. However, on any given machine, some of the `short`, normal and `long` types may be represented in the same way. C also provides a range of pointer types which may, for purposes of interchange with FORTRAN, all be condensed into the generic pointer type `void*`. Note that, unlike FORTRAN, a C character variable can only store a single character. Also unlike FORTRAN, a C character variable is treated as a type of integer rather than as a separate type. Finally, ANSI C has the type `enum`, an enumerated list of values. The aggregate data types are the array, structure and union. New types can be defined in terms of the basic types by means of a `typedef` statement.

When writing mixed language programs, it is clearly important to know which FORTRAN types map on to which C types; in particular, which similar types use the same amount of storage. This is discussed more fully in the machine dependent sections in appendix A; however, there are some general points to be considered first.

3.1 Numeric Types

If data are to be passed between routines that have been written in different languages, then it is important that those languages represent the data in the same way. The FORTRAN standard makes no statements about how any of the data types should be implemented and there is almost nothing in the C standard either. For example, if a certain bit pattern was interpreted as the integer -2 by FORTRAN, yet the same bit pattern was interpreted by C as -1, then there are going to be serious problems trying to communicate between routines written in different languages. Fortunately, the hardware on which the program is running provides a constraint for those data types that are implemented directly in the hardware. For example, all reasonable computers have instructions for operating on integers and it would be a particularly perverse compiler writer who chose not to use the hardware representation. Something that is slightly more likely to be a problem is the way that floating point numbers are represented. If the hardware supports floating point arithmetic, then you are in the same situation as for integers and all should be well. However, if the hardware does not support floating point arithmetic, then there could be problems. Some older PCs do not have floating point hardware, although modern PCs either support floating point operations directly in hardware, or there is a recognised way of representing floating point numbers that is generally adhered to. The bottom line on numerical data types is that it is most unlikely that different languages will represent the same number in a different manner on the same hardware.

3.2 Characters

When considering character data, things are a bit more complicated in that the hardware does not impose a meaning on a given bit pattern. It is the operating system that does that. The character codes that are in common use are the ASCII collating sequence and the EBCDIC collating sequence. EBCDIC is only used by IBM mainframe and minicomputers (and their clones), but there are a lot of IBM computers around. (The IBM PC does *not* use EBCDIC.) Again it would be rather perverse if, on a given computer, FORTRAN and C used a different representation of characters, so that is not really worth worrying about. What certainly is worth paying attention to is the possibility that any given program may be run on several different computers, some using ASCII characters and some using EBCDIC. That is not a concern that is particular to mixed language programming though.

An important point about character data is that they are stored differently in FORTRAN and C. FORTRAN stores character data as a fixed-length string padded with trailing blanks whereas C stores character data as a variable-length, null-terminated string. The difference is standardized, so it does not lead to problems with portability, but it is something that will involve extra work when passing character data between routines written in different languages.

3.3 Logical Types

So far all seems well. However, a place that can certainly cause problems is the representation of logical values. In principle, it is completely up to the compiler writer to chose how logical values are represented. What is even worse as far as C is concerned is that there is no logical type at all! In C, a numerical value of zero represents false and anything else represents true, but these are numeric data types, not logical types. On a VAX/VMS system, FORTRAN represents a logical value of false by an integer zero and true by an integer minus one; however, only the bottom bit is tested, so if an integer value of 2 were to be treated as a logical value, then it would be taken as false. C, on the other hand, would treat it as true.

3.4 Pointer Types

The main reason for passing pointer information between C and FORTRAN is to pass references to dynamically allocated memory, which is especially useful given FORTRAN 77's lack of dynamic memory allocation. In addition, the referenced memory may contain data values which are, in effect, also being exchanged and which we must therefore be able to reference from both languages.

C provides a wide range of pointer types which can be constructed to refer to any other C type. Each of these pointer types can, at least in principle, have different storage requirements. Indeed, on some machines and operating systems there are variations in pointer length according to the type of data being referenced, and even variations in the way bit patterns are interpreted according to where the referenced data are stored in memory. Fortunately, the more arcane of these schemes are now regarded as historical anomalies and are unlikely to be met in future.

C provides the generic pointer type `void*`, to which all pointer types may be cast, and from which the original pointer may later be recovered by casting back to the original type. Since the `void*` type must therefore cater for the "lowest common denominator" of C pointer types, it is very likely to contain just a simple memory address for the referenced data (or something equivalent) on all machines. Therefore, exchanging the `void*` type is the key to interchanging pointers between C and FORTRAN.

However, FORTRAN 77 does not have a pointer type, and its `INTEGER` data type must be pressed into service in order to store pointer values passed from C. Unfortunately, on some platforms, a C pointer is longer than a FORTRAN `INTEGER`, which means that there is no suitable standard (and therefore portable) FORTRAN data type of sufficient length to store an address in memory. To overcome this limitation, some trickery in required, the upshot of which is that there are some restrictions on the particular pointer values which may be passed from C to FORTRAN.

In practice, this means that pointer exchange between C and FORTRAN is really only safe when referring to dynamically allocated memory (and not, for example, when referring to static memory allocated in C, where you have no control over the address used). It also means that CNF must provide special facilities for allocating dynamic memory from C which will later be passed to FORTRAN, and for "registering" the associated pointers. It also provides functions for converting between the C and FORTRAN representations of these pointers.

3.5 Arrays

Although the representation of a single numerical value is unlikely to cause a problem, the way that arrays of numbers are stored is different between different languages. One dimensional arrays are the least problem, but even then there are differences. In C, all arrays subscripts start at zero, and this cannot be changed. In FORTRAN, subscripts start at one by default, but this can be modified so that the lower bound of a dimension of an array can be any integer. What must be remembered is that the array element with the lowest subscript in a FORTRAN array will map on to the array element with a zero subscript when treated as a C array. This is not a serious problem as long as you remember it.

Multi-dimensional arrays are a well known problem since FORTRAN stores consecutive array elements in column-major order (this *is* specified in the FORTRAN standard) whereas other languages store them in row-major order. For example, in FORTRAN, the order of elements in a 2 x 2 array called `A` are `A(1,1), A(2,1), A(1,2), A(2,2)`, whereas in C this would be `A[0][0], A[0][1], A[1][0], A[1][1]`. In practice this is rarely a serious problem as long as you remember to take account of the reversed order when writing a program. However, when coupled with the difference in default lower bounds (zero in C, one in FORTRAN) it is a fruitful source of bugs.

There are additional problems with FORTRAN character arrays. This is because C handles a one dimensional FORTRAN character array as a two dimensional array of type `char`, *i.e.* the FORTRAN statement:

```
      CHARACTER * ( NCHAR ) NAMES(DIM)
```

is equivalent to the C statement:

```
      char names[dim][nchar]
```

3.6 Same Language – Different Compiler

In the preceding sections, reference is often made to "the way that FORTRAN does something" or "the way that C does something". However, even different compilers for the same language can do things in a different way if the standard does not specify how that something should be done. A reasonable example is that one FORTRAN compiler might represent a true logical value by the integer 1, whereas another might just as reasonably use -1. This is not just a hypothetical problem; the FORTRAN for RISC compiler from MIPS and the DEC FORTRAN for RISC compiler both work on the DECstation and interpret the same number as different logical values. I shall continue to refer to "the way that FORTRAN does it", even though it is more correct to refer to "the way that FORTRAN compiler XYZ implements it". The distinction is rarely important, but should be borne in mind.

4 Communication Between Routines

There are three ways of passing data between a program and a subprogram: (i) the argument list of the subprogram, (ii) the return value of the subprogram if it is a function and (iii) global variables. The concept of arguments and return values of subprograms are common to many programming languages including FORTRAN and C. The ways in which global variables are handled are rather different. In FORTRAN there are common blocks whereas in C there are external variables or structures. Each of these will be considered in turn.

4.1 Arguments

This is the main method for passing data between a calling program and the called subprogram. The calling program takes the actual arguments of the call to the subprogram, constructs an argument list and then passes execution to that routine. The subprogram then uses the values in the argument list to access the actual arguments. It may pass data back to the calling program by modifying the data in some or all of the arguments. As far as passing arguments between a program and a subprogram is concerned, the principal difference between FORTRAN and C is the method used for passing the arguments.

Note that the above paragraph refers to modifying arguments. On all the machines we support, the actual contents of the argument list is never modified. What may be modified is the contents of the location pointed to by an element of the argument list. This is not to say that other computers would not modify the argument list itself.

There are three commonly used methods for passing subprogram arguments: call by value, call by reference and call by descriptor. Call by value passes the actual value of the argument to the called routine, call by reference passes a pointer to the value of the argument (*i.e.* the address of the argument) and call by descriptor passes a pointer to a structure describing the argument.

Although these are the basic methods of passing arguments, a particular type of argument may be passed by a combination of these. For example, some compilers use a combination of call by reference and call by value to pass character arguments. What is common is that all arguments are passed by exchanging data values. It is how those values are to be interpreted that gives rise to the different mechanisms.

The FORTRAN standard does not specify how arguments should be passed to subprograms and indeed different compilers for different machines do use different methods. It is most usual for numeric data types to be passed by reference since the subprogram may modify the value of the argument. This is most easily achieved by passing a pointer to the storage location containing the data value, rather than a copy of the value itself. On the other hand, the C standard explicitly states that values cannot be returned to the calling routine directly through arguments and so call by value is most commonly used. It is worth recalling that the argument list of a routine is simply a sequence of computer words. If these are a list of addresses of data values then everything is simple. However, suppose that an array was passed by value. This would mean that the compiler would have to arrange for a copy of the entire array to be placed in the argument list that was passed to the called subprogram. Consequently, arrays are invariably passed by reference or by descriptor, never by value.

It may seem tedious to have to think about the actual mechanisms that a compiler uses to pass data between routines when all you want to do is to get on with your programming. However, understanding this is the key to mixed language programming. Fortunately the facilities described in More on Calling C from FORTRAN (Section 5) and More on Calling FORTRAN from C (Section 8) hide much of this from the programmer.

4.2 Function Values

The second mechanism for passing data between routines is the return value of a function. FORTRAN makes a distinction between subprograms that return a value (functions) and those that do not (subroutines), whereas C does not. All C subprograms are functions that return a value (even the main program), although that value may be `void`. Since it is simply a value that is being returned, the mechanisms for returning scalar numeric values tend to be just that – a value is returned. However, things get more complicated in the case of functions returning things like character variables. This will be discussed further in appendix A on machine dependencies.

4.3 Global Data

Different languages can have very different ways of dealing with variables that are not local to a particular routine, but have a more global scope. FORTRAN has common blocks for global data that are accessed by particular routines. The data values in a common block can be accessed by different names in different routines, although this is generally considered bad practice. C functions can access global data by using variables that are not declared in a particular routine, but have a scope of all the routines contained in the source file in which the global variables are defined. If the same variable is needed across several source files, then it can be declared as `extern`.

Although these two mechanisms are very different in principle, in practice, computer manufacturers tend to implement them in a way such that it is possible to share global data between routines that have been written in different languages. The details of how this is done are given in the appendix about specific machines. However, there is an indirect way of accessing FORTRAN common blocks from C that is also worth considering. The FORTRAN routine that calls the C function can pass as an argument, the first element of the common block. As long as FORTRAN passes this argument by reference, then the C function can use this address to access all of the other elements of the common block. The elements of the common block must be stored contiguously. Whether this method, or the use of the F77 macros (described in Section 5.8), achieve a greater degree of portability in this respect is not known at present. On account of these potential portability problems, you should avoid passing global data between routines written in different languages, whenever possible.

5 More on Calling C from FORTRAN

As the examples in the appendix on machine specific details show, different computers handle subroutine interfaces in different ways. This apparently makes it difficult to write portable programs that are a mixture of FORTRAN and C. However, it is only the C code that differs and fortunately the differences can be hidden by suitable C macros so that the same code can be compiled on all types of hardware mentioned in this document. The macros have been constructed in such a way that they can accommodate other subroutine passing mechanisms; however, it is not possible to guess all the types of mechanisms that we might come across.

The macros can be used in a C function by including the file `f77.h`. This file will naturally be stored in different places on different types of system, even if it is only the syntax of the file name that is different. It would be a pity if all of the implementation specific details were hidden away in these macros, only to have to have an implementation specific `#include` statement in each C source file. Fortunately there is a way around this problem that is described in Section 13 on compiling and linking.

Let us now consider an example of using the F77 macros to illustrate their use. The following example generates a banner which consists of some hyphens, followed by some stars and finally the same number of hyphens again. There are also some blanks between the beginning of the line and between the hyphens and stars. The work is done in the subroutine BANNER and the form of the output is governed by the three arguments FIRST, MIDDLE and GAP. For example, `CALL BANNER( LINE, 5, 10, 3 )` would return with `LINE` set to the following character string.

```
   -----   **********   -----
```

Example 3 – Passing arguments between FORTRAN and C.

FORTRAN program:

```
      PROGRAM F1
      INTEGER FIRST, MIDDLE, GAP
      CHARACTER*(80) LINE

      FIRST = 5
      MIDDLE = 10
      GAP = 3
      CALL BANNER( LINE, FIRST, MIDDLE, GAP )
      PRINT *, LINE

      END
```

C function:

```
#include "f77.h"

F77_SUBROUTINE(banner)( CHARACTER(line), INTEGER(first), INTEGER(middle),
                       INTEGER(gap) TRAIL(line) )  {
  GENPTR_CHARACTER(line)
  GENPTR_INTEGER(first)
  GENPTR_INTEGER(middle)
  GENPTR_INTEGER(gap)
  int i, j;       /* Loop counters.  */
  char *cp;       /* Pointer to a character.  */

/* Make cp point to the beginning of the string line.  */
  cp = line;

/* First blanks.  */
  for( i = 0, j = 0 ; (j < line_length) && (i < *gap) ; i++, j++ )
     *cp++ = ' ';

/* First hyphens.  */
  for( i = 0 ; (j < line_length) && (i < *first) ; i++, j++ )
     *cp++ = '-';

/* More blanks.  */
  for( i = 0 ; (j < line_length) && (i < *gap) ; i++, j++ )
     *cp++ = ' ';

/* Middle stars.  */
  for( i = 0 ; (j < line_length) && (i < *middle) ; i++, j++ )
     *cp++ = '*';

/* More blanks.  */
  for( i = 0 ; (j < line_length) && (i < *gap) ; i++, j++ )
     *cp++ = ' ';

/* Last hyphens.  */
  for( i = 0 ; (j < line_length) && (i < *first) ; i++, j++ )
```

```
            *cp++ = '-';
      }
```

The FORTRAN part of this example is completely standard; it is the C code that need further explanation. Firstly there is the declaration of the subroutine with the macro F77_SUBROUTINE. This handles the fact that some computer systems require a trailing underscore to be added to the name of the C function if it is to be called from FORTRAN. In the same statement there are the function's dummy arguments, declared using the macros CHARACTER, INTEGER and TRAIL. The INTEGER macro declares the appropriate dummy argument to handle an incoming argument passed from FORTRAN. This will usually be declared to be "pointer to int". The CHARACTER and TRAIL macros come in pairs. The CHARACTER macro declares the appropriate argument to handle the incoming character variable and TRAIL may declare an extra argument to handle those cases where an extra hidden argument is added to specify the length of a character argument. On some machines, TRAIL will be null and on account of this *there should not be a comma before any TRAIL macros*. When TRAIL is not null, then it will add the comma itself. If there are several TRAIL macros then there must not be a comma directly in front of any of them.

The next set of macros are the GENPTR_*type* macros, one for each argument of the FORTRAN subroutine (TRAIL arguments are not counted as separate arguments for this purpose). These handle the ways that subprogram arguments may be passed on different machines. They ensure that a pointer to the argument exists. On most systems, this is exactly what is passed from the FORTRAN program and so the macros for numeric arguments are null. If a particular system passed the value of an argument, rather than its address, then these macros would generate the appropriate pointers.

The CHARACTER, TRAIL and GENPTR_CHARACTER macros have to cope with the different ways that systems deal with passing character variables. Although the way that these macros are implemented can be a bit complex, what the programmer sees is essentially simple. For each character argument, the macros generate a pointer to a character variable and an integer holding the length of that character variable. The above example will create the variable line of type char * and variable line_length of type int. If these are available directly as function arguments, then the macro GENPTR_CHARACTER will be null, otherwise it will generate these two variables from the arguments. The best way of seeing what is going on is to compile a function with macro expansion turned on and list the output.

There is an important difference between this example and the one in the cookbook. In this case, an int variable containing the length of the character argument is generated automatically whereas in the example in the cookbook the length was passed explicitly. In fact, the int variable was also generated in the example in the cookbook, but it was not used. It is more portable to explicitly pass the length of CHARACTER variables and to ignore the automatically generated length as this will cope with the situation where the length cannot be generated automatically. No such machines are known to the author at present, but Murphy's Law would indicate that the next machine that we desperately need to use will have this problem.

Although the use of these macros does seems a bit strange at first, once any pointers have been generated, the rest of the code is standard C.

Something that has not yet been considered is whether to write the code in upper or lower case. All of the examples in this document have the FORTRAN code in upper case and the C code in lower case, thereby following common practice. Normally it makes no difference whether code is written in upper case or lower case. Where it does matter is in declaring external symbols. External symbols are names of routines and names of common blocks (FORTRAN) or variables declared `extern` (C). The linker must be able to recognise that the external symbols in the FORTRAN routines are the same external symbols in the C functions. On a VMS system, the VAX C compiler will fold all external symbols to upper case by default, although there is a compiler option to fold them all to lower case or leave them as written in the source code. The VAX FORTRAN compiler will generate all external symbols in upper case. On Unix systems, the FORTRAN compiler will typically fold external names to lower case (and add a trailing underscore), whereas the C compiler will leave the case unchanged. Consequently, all external symbols in C functions that might be referenced from FORTRAN should be coded in lower case.

5.1 General Description

Having considered an example of using the macros to write a C function that is to be called from FORTRAN, let us look at all of the macros in more details. You will notice that some of the macros are prefixed by F77 while others are not. Those that do not have the F77 prefix are those that occur in standard places in the source code and so are unlikely to be confused with other macros. The macros that do have the F77 prefix are those that declare a C function and others that are less commonly used, and when they are, they can occur anywhere within the body of the C routine. A full description of each macro is available in appendix E.

The whole ethos of the F77 macros is to try to isolate the FORTRAN/C interface to the beginning of the C function. Within the body of the C function, the programmer should not need to be aware of the fact that this function is designed to be called from FORTRAN. It is not possible to achieve this completely and at the same time retain portability of code, but the intention is there none the less.

5.2 Declaration of a Function

There are two types of macros involved in declaring a C function that is to be called from FORTRAN; the function name and the function arguments. If the C function is to be treated as a FORTRAN subroutine, then it should be declared with the macro `F77_SUBROUTINE`. This will declare the C function to be of type `void` and will generate the correct form of the of the routine name, handling such things as appending a trailing underscore where required.

If the C function is to be treated as a FORTRAN numerical or logical function, then it should be declared with one of the macros `F77_`*`type`*`_FUNCTION`. These macros will declare the function to be of the appropriate type, *e.g.* a function declared with `F77_INTEGER_FUNCTION` is likely to be of type `int`.

The declaration of a C function that is to be treated as a FORTRAN character function is more complex than one that returns a scalar numeric or logical value. The first argument of the function should be `CHARACTER_RETURN_VALUE`(return_value), where `return_value` is a variable of type “pointer to `char`”. Although character functions work perfectly well on all current Starlink hardware, it is one of the more difficult things to guess how other manufacturer might implement them. Consequently, it is recommended that character functions be avoided where possible and that a subroutine that returns a character argument be used instead.

5.3 Declaration of Arguments

Scalar arguments are declared with the macros `INTEGER`, `REAL`, `DOUBLE`, `LOGICAL`, `CHARACTER` and `TRAIL`. (Or the non-standard `BYTE`, `WORD`, `UBYTE`, `UWORD` or `POINTER`.) The macros that declare numeric and logical arguments take account of the fact that a FORTRAN integer variable may correspond to a C type of `int` on one machine, but to `long int` on another. They also handle the mechanism that is used to pass the arguments.

Character arguments are more complex as different computers use differing mechanisms for passing the arguments. To take account of this, for every argument that is declared using the `CHARACTER` (or `CHARACTER_ARRAY`) macro, there should be a corresponding `TRAIL` macro at the end of the list of dummy arguments. As mentioned in a preceding example, *there should not be a comma before any `TRAIL` macros*.

C differs from FORTRAN in that it has pointer variables. These are often used to manipulate arrays, rather than by using array subscripts. The macros that are used to declare array arguments do in fact declare them to be arrays. If programmers wish to manipulate these arrays by means of pointer arithmetic, then for maximum portability they should declare separate pointers within the C function that point to the array argument.

Array arguments are declared by one of the macros *`type`*`_ARRAY`. The macros that declare numeric or logical array arguments declare the arrays to be pointers to *type*. To enable the C function to process the array correctly, the dimensions of the array should be passed as additional arguments.

The F77 macros do not allow you to declare fixed sized dimensions for an array that is a dummy argument. Normally, it is necessary to pass the dimensions as arguments of the routine anyway, but there are circumstances where the dimensions of the array will be fixed, *e.g.* an array might specify a rotation in space and hence is always 3 x 3. What is gained by declaring the fixed dimensions of the array is that subscript calculations can be done on arrays of more than one dimension. Unfortunately, such declarations cannot be made portable as some FORTRAN systems pass arrays by descriptor. If you really must declare arrays with fixed dimensions, you can do so as follows:

```
F77_SUBROUTINE(subname)( F77_INTEGER_TYPE array[3][3] )
{
  ...
  elem = array[i][j]
  ...
}
```

This example declares the dummy argument to be an INTEGER array of fixed size. Although the subscript calculation can be performed as the routine knows the size of the array, the `sizeof` operator does not return the full size of the array as the complier casts `array[3][3]` to `*array`. All things considered, it is better to have the dimensions of arrays passed as separate arguments and to do the subscript arithmetic yourself with pointers. Here is an example of initializing an array of arbitrary size and arbitrary number of dimensions.

Example 4 – Passing an array of arbitrary size from FORTRAN to C.

FORTRAN program:

```
      PROGRAM ARY

      INTEGER NDIMS, DIM1, DIM2, DIM3
      PARAMETER( NDIMS = 3, DIM1 = 5, DIM2 = 10, DIM3 = 2 )

      INTEGER DIMS( NDIMS )
      INTEGER A( DIM1, DIM2, DIM3 )

      DIMS( 1 ) = DIM1
      DIMS( 2 ) = DIM2
      DIMS( 3 ) = DIM3
      CALL INIT( A, NDIMS, DIMS )

      END
```

C function:

```
#include "f77.h"

F77_SUBROUTINE(init)( INTEGER_ARRAY(a), INTEGER(ndims), INTEGER_ARRAY(dims) )
{
  GENPTR_INTEGER_ARRAY(a)
  GENPTR_INTEGER(ndims)
  GENPTR_INTEGER_ARRAY(dims)

  int *ptr = &a[0];  /* ptr now points to the first element of a.  */
  int size = 1;      /* Declare and initialize size.  */
  int i;             /* A loop counter.  */

  /* Find the number of elements in a.  */

  for( i = 0; i < *ndims ; i++ )
     size = size * dims[i];

  /* Set each element of a to zero.  */

  for( i = 0 ; i < size ; i++ )
     *ptr++ = 0;
}
```

In this example, each element of the array `a` is accessed via the pointer `ptr`, which is incremented each time around the last loop.

5.4 Arguments – and Pointers to Them

When a FORTRAN program calls a subprogram, it is possible for the value of any of its arguments to be altered by that subprogram. In the case of C, a function cannot return modified values of arguments to the calling routine if what is passed is the value of the argument. If a C function is to modify one of its argument, then the address must be a pointer to the value to be modified rather than the actual value. Consequently in C functions that are designed to be called from FORTRAN, all function arguments should be treated as though the address of the actual argument had been passed, not its value. This means that the arguments should be referenced as `*arg` from within the C function and not directly as *arg*. This may seem odd to a FORTRAN programmer, but is natural to a C programmer.

To ensure that there always exists a pointer to each dummy argument, the first lines of code in the body of any C function that is to be called from FORTRAN should be `GENPTR` macros for each of the function arguments. The macros `GENPTR_type` always result in there being a C variable of type "pointer to type" for all non-character variables. For example, `GENPTR_INTEGER(first)` ensures that there will be a variable declared as `int *first`. On all current types of system, this macro will actually be null since the pointer is available directly as an argument. However, the macro should be present to guard against future computers working in a different way. For example, if a particular system passed FORTRAN variables by value rather than by reference, then this macro would construct the appropriate pointer.

Character arguments are different in that the `GENPTR` macro ensures that there are two variables available, one of type "pointer to char" that points to the actual character data, and one of type `int` that is the length of the character variable. The name of the variable that holds the length of the character string is constructed by appending "`_length`" to the name of the character variable. For example, if a function is declared to have a dummy argument with the macro `CHARACTER(ch)` and a corresponding `TRAIL(ch)`, then after the execution of whatever the macro `GENPTR_CHARACTER(ch)` expands into, there will be a "pointer to character" variable called `ch` and an integer variable called `ch_length`. Although the length of a character variable is directly accessible through the `int` variable `ch_length`, it is better to pass the length of the character variable explicitly if maximum portability is sought. This is because, although it works on all currently supported platforms, it may not be possible to gain access to the length on some machines.

It is important to remember that what is available after the execution of what a `GENPTR` macro expands into will be a pointer to the dummy argument, not a variable of numeric or character type. Consequently the body of the code should refer to it as `*arg` and not as `arg`. In a long C function, it may be worth copying scalar arguments into local variables to avoid having to remember to put the `*` on each reference to an argument. If the variable is changed in the function, then it should of course be copied back into the argument at the end of the function. Alternatively you could define C macros to refer to the pointers, such as

```
#define STATUS *status
```

Note that although ANSI C will allow the above as `status` and `STATUS` are distinct names, you should beware of the possibility of a computer that does not have lower case characters. Such machines used to exist in abundance, but at present, this does seem a remote possibility.

Array arguments should have pointers generated (if necessary) by using the `GENPTR_type_ARRAY` macros. All arrays are handled by these macros.

5.5 Type Specifiers

There are macros `F77_type_TYPE` which expand to the C data type that corresponds to the FORTRAN data type of the macro name, *e.g.* on a particular computer `F77_INTEGER_TYPE` may expand to `int`. These are usually not needed explicitly within user written code, but can be required when declaring common blocks, casting values from a variable of one type to one of a different type and when using the `sizeof` operator.

5.6 Logical Values

The macros `F77_FALSE` and `F77_TRUE` expand to the numerical values that FORTRAN treats as false and true (*e.g.* 0 and 1). They should be used when setting logical values to be returned to the calling FORTRAN routine. There are also macros `F77_ISFALSE` and `F77_ISTRUE` that should be used when testing a function argument for truth or falsehood.

5.7 External Names

The macro `F77_EXTERNAL_NAME` handles the difference between the actual external name of a function called from FORTRAN and a function that apparently has the same name when called from C. Typically this involves appending an underscore character to a name. This macro is not normally needed directly by the programmer, but is called by other macros.

5.8 Common Blocks

There are two macros that deal with common blocks, `F77_NAMED_COMMON` and `F77_BLANK_COMMON`. They are used when declaring external structures that corresponds to FORTRAN common blocks and when referring to components of those structures in the C code. The following declares a common block named "block" that contains three `INTEGER` variables and three `REAL` variables.

```
extern struct
{
 F77_INTEGER_TYPE i,j,k;
 F77_REAL_TYPE a,b,c;
} F77_NAMED_COMMON(block);
```

The corresponding FORTRAN statements are

```
      INTEGER I,J,K
      REAL A,B,C
      COMMON /BLOCK/ I,J,K,A,B,C
```

Within the C function the variables would be referred to as:

`F77_NAMED_COMMON(block).i`, `F77_NAMED_COMMON(block).j`, *etc.*

Note that all that these macros do is to hide the actual name of the external structure from the programmer. If a computer implemented the correspondence between FORTRAN common blocks and C global data in a completely different way, then these macros would not provide portability to such an environment.

On account of this, it is best to avoid using common blocks where possible, but of course, if you need to interface to existing FORTRAN programs, this may not be practical.

6 Converting Between FORTRAN and C Strings

6.1 The CNF Functions

FORTRAN stores `CHARACTER` strings as fixed-length strings filled with trailing blanks, whereas C stores them as a variable-length strings each terminated by the null character. Although C strings are of variable length, there must of course be enough space reserved to store the maximum length that the string ever reaches *plus one more character for the trailing null*.

To aid the programmer in converting between the two forms of character strings, a number of C functions are provided in the CNF library. These handle all aspects of converting between the two types of string and provide options such as creating temporary strings, including the trailing blanks in the C version of a string and only copying a maximum number of characters. The process of converting from FORTRAN to C strings is known as "importing" and from C to FORTRAN as "exporting".

None of the functions are very complicated and some of them are just a tidier way of achieving what could be done with a few lines of C in the calling program. Consequently in a time critical application it may be appropriate to include the source of a CNF function in your code, rather than incur the overheads of a making a function call.

Full descriptions of the CNF functions are provided in Appendix G.

Here is an example of how to use them. This is the same as an example from the machine specific section of this document. The use of the F77 macros and the CNF functions have made the C code easier to write and completely portable to all Starlink systems.

Example 5 – Converting character arguments between FORTRAN and C.

FORTRAN program:

```
      PROGRAM STRING
      CHARACTER STR*20

      CALL GETSTR( STR )
      PRINT *,STR

      END
```

C function:

```
#include "f77.h"

F77_SUBROUTINE(getstr)( CHARACTER(fortchar) TRAIL(fortchar) )
{
  GENPTR_CHARACTER(fortchar)           /* Generate pointer to fortchar */

  char  *string = "This is a string";  /* A string to be printed */

/* Copy the string to the function argument */
  cnfExprt( string, fortchar, fortchar_length );
}
```

Other examples in this document illustrate the use of CNF functions for importing strings and calling FORTRAN from C.

6.2 Handling Byte Strings (HDS Locators)

Sometimes FORTRAN `CHARACTER` variables are used to contain strings of bytes rather than normal, printable character strings – a particular case of this is HDS locators (see SUN/92). In this case, special characters, such as NULL, cease to have their normal meaning and this could confuse the standard CNF import and export functions. For this reason, functions `cnfImpch` and `cnfExpch` are provided. These functions just import and export a given number of characters.

6.3 Using Dynamic FORTRAN Character Strings

The `DECLARE_CHARACTER` macro used in an earlier example (2) assumes that the length of the required FORTRAN character string is a constant, known at compile time. This is not always the case – for example, the character argument to be passed to the FORTRAN subroutine may be derived from an argument of the calling C function as in the case of a C wrap-around for a FORTRAN subroutine. To cater for this situation, macros are provided which will allocate and free space for the FORTRAN character string at run time. They make use of the CNF functions `cnfCref` and `cnfFreef`.

The following example illustrates their use for both input and output of strings from a FORTRAN subroutine which takes a given string, modifies it and returns the result.

Example 6 – Dynamic CHARACTER Arguments.

C main program

```
void strStrip( char *in, char *out, int maxout );

main(){
char in[20]="Hello  there  !";
char out[20];

printf( "Input string is: %s\n", in );
strStrip( in, out, 20 );
printf( "Output string is: %s.\n", out );
}
```

C wrap-around for a FORTRAN subroutine

```
/* strStrip - A C wrap-around for FORTRAN subroutine STR_STRIP */
#include "f77.h"

extern F77_SUBROUTINE(str_strip)
  ( CHARACTER(fin), CHARACTER(fout) TRAIL(fin) TRAIL(fout) );

void strStrip( char *in, char *out, int maxout ){
  DECLARE_CHARACTER_DYN(fin);
  DECLARE_CHARACTER_DYN(fout);
```

```
     F77_CREATE_CHARACTER(fin,strlen(in));
     F77_CREATE_CHARACTER(fout,maxout-1);

     cnfExprt( in, fin, fin_length );

     F77_CALL(str_strip)
        ( CHARACTER_ARG(fin), CHARACTER_ARG(fout)
          TRAIL_ARG(fin) TRAIL_ARG(fout) );

     cnfImprt( fout, fout_length, out );

     F77_FREE_CHARACTER(fin);
     F77_FREE_CHARACTER(fout);
   }
```

which is a C wrapper for the FORTRAN subroutine:

```
         SUBROUTINE STR_STRIP( FIN, FOUT )
   * Remove multiple spaces from a string
         IMPLICIT NONE
         INTEGER I, J
         CHARACTER*(*) FIN
         CHARACTER*(*) FOUT

         FOUT = FIN(1:1)
         I = 2
         J = 1

         DOWHILE ( I .LE. LEN(FIN) )
            IF ( FIN(I:I) .NE. ' ' ) THEN
               J = J + 1
               FOUT(J:J) = FIN(I:I)
            ELSE IF ( FOUT(J:J) .NE. ' ' )
               J = J + 1
               FOUT(J:J) = FIN(I:I)
            END IF

            I = I + 1

         ENDDO

         END
```

Here, `DECLARE_CHARACTER_DYN` is used in place of `DECLARE_CHARACTER`. It declares pointers rather than allocating space for the FORTRAN character strings to be passed to the FORTRAN subroutine. A variable to hold the string length is also declared.

The `F77_CREATE_CHARACTER` expands to executable statements which allocate space and set the pointers and string length. The `F77_FREE_CHARACTER` macro expands to executable statements which free the previously allocated space.

7 Pointers

FORTRAN 77 does not have the concept of a pointer. However, FORTRAN `INTEGER`s are widely used in Starlink software as a replacement for pointers when passing the address of a data array from one routine to another. Typically, a FORTRAN program calls a subroutine that returns a value in an `INTEGER` variable that represents the address of an array, which will usually have been dynamically allocated. The value of this variable (as opposed to its address) is then passed on to another routine where the contents of the array are accessed.

C, of course, does provide pointers – in fact you can hardly avoid using them – and they are distinct from C integers (`int`s). To take account of this, a macro `POINTER` is defined to declare C function arguments that the calling FORTRAN program declares as an `INTEGER` but will actually treat as a pointer. The FORTRAN routine should not process a `POINTER` variable in any way. The only valid operations it may perform are to copy it, to pass it to a subprogram using the normal parameter passing mechanism, or to pass its value to a subprogram using the `%VAL` facility (as described in Section 7.3) in order to access the contents of the array to which it points.

Unfortunately, this scheme of using FORTRAN `INTEGER`s to hold pointer values only works cleanly if the length of an `INTEGER` is the same as the length of the C generic pointer type `void*`. Where this is not the case (on DEC Alphas for instance), some way around the problem has to be found.

On some systems, the linker may have flags to control the size of the address space in which a program runs, and this can provide a simple solution. For example, although addresses on DEC Alphas are normally 64 bits long (the length of a C pointer), it is possible to force programs to use addresses in which only the lowest 32 bits have non-zero values. It then becomes a simple matter to convert C pointers into FORTRAN `INTEGER`s (which are 32 bits long) because discarding the most significant bits has no effect. The standard Starlink link scripts on DEC Alpha systems supply the necessary command-line flags to produce this behaviour automatically.

However, this simple solution is not always applicable. Apart from the possibility that future 64-bit operating systems (of which there are likely to be an increasing number) may not provide this option of running programs in “lower memory”, even on those that do the option cannot always be exercised. For example, some software packages make use of “dynamic loading” of routines stored in shareable libraries as a way of allowing their capabilities to be extended. This is a very flexible facility, but it means that the loaded routines must execute in the address space of the main program, which means that the writer of the shareable library no longer has any control over the number of bits used in pointers. The only option is then to re-build the main software package with the required linker options. This is at best inconvenient, but in the case of commercial software packages it may be impossible.

There is also an increasing likelihood that programs may need to access data arrays of such size that 32 bits of address space (the usual length of FORTRAN `INTEGER`s) is insufficient.

7.1 Pointer Registration and Conversion

To overcome these problems, some method is needed of converting between (say) 64-bit C pointers and the typical 32 bits of a FORTRAN `INTEGER`. The same method must also work if the two pointer representations are actually of equal length. To allow this, CNF maintains an internal table which contains all the C pointers which will be exported and used from FORTRAN. The pointers stored in this table are said to have been “registered” for use from both C and FORTRAN.

When converting a C pointer into a FORTRAN pointer, it is sufficient simply to mask out all bits except those that will fit into a FORTRAN `INTEGER`. This is performed by the function `cnfFptr`. When converting in the opposite direction, the internal table must be searched to locate a pointer which has the same value stored in the set of masked bits (e.g. the lowest 32 bits) as the FORTRAN pointer value. The full value of the C pointer can then be read from the table. This conversion is performed by `cnfCptr`.

Apart from the requirement that all pointers which will be used from both C and FORTRAN must be registered by entering them in the internal table, this scheme also requires that all registered pointers should be unique in their lowest 32 bits (or whatever length a FORTRAN `INTEGER` has) in order for the conversion from FORTRAN to C to select a unique pointer from the table. In practice, these requirements are most easily fulfilled by providing a set of memory allocation functions in CNF which mirror the standard C run time library functions `malloc`, `calloc` and `free`.

7.2 Allocating Exportable Dynamic Memory

The CNF functions `cnfMalloc` and `cnfCalloc` should be used whenever you wish to dynamically allocate memory in a C function and export the resulting pointer for use from FORTRAN. You might also want to use them if you are writing a subroutine library that returns pointers to dynamic memory through its public interface, since the caller might then decide to pass these pointers on to a FORTRAN routine.

For example, here is how you should allocate space for an array of `N` FORTRAN `REAL` values in a C function and pass back the resulting pointer to FORTRAN:

```
F77_POINTER_FUNCTION(ralloc)( INTEGER(N) )
{
   GENPTR_INTEGER(N)

/* Allocate the memory and return the converted pointer. */
   return cnfFptr(cnfMalloc(*N*sizeof(F77_REAL_TYPE)));
}
```

When the allocated memory is no longer required, it should be freed using `cnfFree`. This is how you might import the FORTRAN pointer value allocated above back into C in order to free it:

```
F77_SUBROUTINE(rfree)( POINTER(FPNTR) )
{
   GENPTR_POINTER(FPNTR)

/* Convert back to a C pointer and then free it. */
   cnfFree(cnfCptr(*FPNTR));
}
```

Externally, these CNF memory allocation functions behave exactly like their standard C equivalents `malloc`, `calloc` and `free`. Internally, however, they perform two important additional functions:

- They maintain the internal table of "registered" pointers, so that the conversion functions cnfFptr and cnfCptr can operate (if you use malloc to obtain a pointer, for instance, then these conversion functions will fail and return zero).
- They ensure that all memory allocation results in pointers whose lowest 32 bits (or the length of a FORTRAN INTEGER) are unique, so that conversion between FORTRAN and C pointer values is a well-defined operation.

For convenience, cnfFree is also able to free pointers which have not been registered, in which case it behaves exactly like free.

7.3 Accessing Dynamic Memory from C and FORTRAN

Of course, exchanging pointers to dynamic memory between C and FORTRAN is only part of the story. We must also be able to access the memory from both languages.

When importing a FORTRAN pointer into C, the first step is to use cnfCptr to convert it to a C pointer of type void*. You can then use a cast to convert to the appropriate C pointer type (which you must know in advance) in order to access the values stored in the memory. For example, to print out the contents of a dynamically allocated array of FORTRAN REAL data from a C function, you might use the following:

```
F77_SUBROUTINE(rprint)( INTEGER(N), POINTER(FPNTR) )
{
   GENPTR_INTEGER(N)
   GENPTR_POINTER(FPNTR)
   F77_REAL_TYPE *cpntr;

/* Convert to a C pointer of the required type. */
   cpntr=(F77_REAL_TYPE)cnfCptr(*FPNTR);

/* Access the data. */
   for(i=0;i<*N;i++) printf("%g\n",cpntr[i]);
}
```

Accessing dynamically allocated memory via a pointer from FORTRAN requires two steps. First, the pointer value stored in a FORTRAN INTEGER must be expanded to its full value (if necessary), equivalent to the full equivalent C pointer. This value must then be turned into a FORTRAN array which can be accessed. This requires that the pointer be passed to a separate FORTRAN routine using the %VAL facility.[1] For example, to convert a pointer into a REAL array, you might call an auxiliary routine RWRITE as follows:

```
      INCLUDE 'CNF_PAR'

      ...

      CALL RWRITE(N,%VAL(CNF_PVAL(PNTR)))
```

[1]This is a non-standard facility which originated in VAX FORTRAN but is now available on most FORTRAN compilers. It is a compiler directive, rather than a function, and works by instructing the compiler to pass the argument by value rather than by address. The routine receiving this argument is then tricked into thinking that it has received an array starting at the address given by the pointer value. There are other ways of achieving this effect, such as by addressing an array outside of its bounds, but the %VAL method is the one most widely used in Starlink software.

and the RWRITE routine could then access the array of values as follows:

```
      SUBROUTINE RWRITE( N, RDATA )
      INTEGER I, N
      REAL RDATA( N )
      DO 1 I = 1, N
         WRITE(*,*) RDATA(I)
    1 CONTINUE
      END
```

Note how the argument of the %VAL directive is CNF_PVAL(PNTR). The FORTRAN-callable CNF_PVAL function serves to expand the pointer value out to its full length (equivalent to calling cnfCptr from C). The data type returned by this function will depend on the length of C pointers on the machine being used and may not be a standard FORTRAN type (for instance, on DEC Alphas it is an INTEGER*8 function). However, the data type declaration for this function is encapsulated in the CNF_PAR include file, so you need not include non-standard type declarations directly in your own software.

7.4 Registering Your Own Pointers

CNF also provides two functions, cnfRegp and cnfUregp, for registering and un-registering pointers – *i.e.* for entering and removing them from the internal table which is used for pointer conversion between C and FORTRAN. You will probably never need to use these, since pointer registration is normally managed completely automatically by the memory allocation functions which CNF provides.

The reason for providing them is that there may be ways of creating new memory, for which cnfMalloc or cnfCalloc cannot be used. For example, mapping data files directly into memory. If the resulting C pointers are to be exported to FORTRAN, they must be accessible to CNF for conversion purposes, so they must be registered in CNF's internal table.

If you should ever need to use this facility, then the main point to note is that attempting to register a C pointer can potentially fail (cnfRegp returns -1 to indicate this). This will occur if, when the C pointer is converted to a FORTRAN INTEGER, it clashes with a FORTRAN pointer value which is already in use. In such a case you cannot safely export your pointer to FORTRAN, so you must obtain a new pointer and re-register it. Typically, this may involve allocating a new block of memory at a different location and freeing the original. The consolation is that such clashes are extremely rare.

8 More on Calling FORTRAN from C

The operations needed to write a C routine that can call a FORTRAN subroutine or function are fairly similar to those needed when calling C from FORTRAN. Many of the macros that are used are the same, so you should read

Section 5 before reading this.

A typical reason to call FORTRAN from C is to use a pre-existing subroutine library. Here is an example of calling PGPLOT from a C main program.

Example 7 – Passing arguments from C to FORTRAN.

C main program:

```
#include "f77.h"

extern F77_SUBROUTINE(pgbegin)
  ( INTEGER(unit), CHARACTER(file), INTEGER(nxsub), INTEGER(nysub)
    TRAIL(file) );

extern F77_SUBROUTINE(pgenv)
  ( REAL(xmin), REAL(xmax), REAL(ymin), REAL(ymax), INTEGER(just),
    INTEGER(axis) );

extern F77_SUBROUTINE(pglabel)
  ( CHARACTER(xlab), CHARACTER(ylab), CHARACTER(toplab)
    TRAIL(xlab) TRAIL(ylab) TRAIL(toplab) );

extern F77_SUBROUTINE(pgpoint)
  ( INTEGER(n), REAL_ARRAY(xs), REAL_ARRAY(ys), INTEGER(symbol) );

extern F77_SUBROUTINE(pgend) ( );

extern F77_SUBROUTINE(pgline)
  ( INTEGER(n), REAL_ARRAY(xpnts), REAL_ARRAY(ypnts) );

main()
{
  int i;
  float xs[] = {1.,2.,3.,4.,5.};
  float ys[] = {1.,4.,9.,16.,25.};

  DECLARE_INTEGER(unit);
  DECLARE_CHARACTER(file,10);
  DECLARE_INTEGER(nxsub);
  DECLARE_INTEGER(nysub);
  DECLARE_REAL(xmin);
  DECLARE_REAL(xmax);
  DECLARE_REAL(ymin);
  DECLARE_REAL(ymax);
  DECLARE_INTEGER(just);
  DECLARE_INTEGER(axis);
  DECLARE_CHARACTER(xlab,50);
  DECLARE_CHARACTER(ylab,50);
  DECLARE_CHARACTER(toplab,50);
  DECLARE_INTEGER(n);
  DECLARE_REAL_ARRAY(xpnts,60);
  DECLARE_REAL_ARRAY(ypnts,60);
  DECLARE_INTEGER(symbol);


  unit = 0; cnfExprt( "?", file, file_length); nxsub = 1; nysub = 1;
  F77_CALL(pgbegin) ( INTEGER_ARG(&unit), CHARACTER_ARG(file),
                      INTEGER_ARG(&nxsub), INTEGER_ARG(&nysub)
                      TRAIL_ARG(file) );

  xmin = 0.0; xmax = 10.0; ymin = 0.0; ymax = 20.0; just = 0; axis = 1;
  F77_CALL(pgenv) ( REAL_ARG(&xmin), REAL_ARG(&xmax), REAL_ARG(&ymin),
                    REAL_ARG(&ymax), INTEGER_ARG(&just), INTEGER_ARG(&axis) );

  cnfExprt( "(x)", xlab, xlab_length );
  cnfExprt( "(y)", ylab, ylab_length );
  cnfExprt( "PGPLOT Example 1 - y = x\\u2", toplab, toplab_length );
  F77_CALL(pglabel) ( CHARACTER_ARG(xlab), CHARACTER_ARG(ylab),
                      CHARACTER_ARG(toplab)
                      TRAIL_ARG(xlab) TRAIL_ARG(ylab) TRAIL_ARG(toplab) );

  n = 5;
  for( i=0 ; i<n ; i++ )
  {
     xpnts[i] = xs[i];
     ypnts[i] = ys[i];
  }
  symbol = 9;
  F77_CALL(pgpoint) ( INTEGER_ARG(&n), REAL_ARRAY_ARG(xpnts),
                      REAL_ARRAY_ARG(ypnts), INTEGER_ARG(&symbol) );

  n = 60;
  for( i=0 ; i<n ; i++ )
  {
     xpnts[i] = 0.1 * i;
     ypnts[i] = xpnts[i]*xpnts[i];
  }
  F77_CALL(pgline) ( INTEGER_ARG(&n), REAL_ARRAY_ARG(xpnts),
                     REAL_ARRAY_ARG(ypnts) );

  F77_CALL(pgend)();

}
```

This is a realistic example of calling PGPLOT routines from C. The module begins with a set of function prototypes for the FORTRAN routines that will be called in the C main program. All variables that need to be passed to FORTRAN subroutines are declared using `DECLARE_`*type* macros. These macros ensure that the variables are declared to be of the correct type and storage size expected by the FORTRAN subroutine. There then follow the calls to the subroutines that do the actual plotting. The most notable things about these calls is that the actual arguments are explicitly passed by address. This seems strange to a FORTRAN programmer, but is natural to a C programmer. Arguments that may be modified must always have their addresses passed, not their values. It may be thought that the *type*`_ARG` macros should add the `&` character where it is needed. However, this gives rise to problems when calling FORTRAN from C from FORTRAN, as well as being rather misleading. Note that scalar arguments need the ampersand character adding, whereas array arguments do not. This is exactly what would be typed if the called routine were a C function.

What is clear from this example is that the inability to put arguments that are constant expressions directly in the call to the routine makes the program a lot more verbose than the equivalent FORTRAN program. Unfortunately, the obvious solution of writing an actual argument as something like INTEGER_ARG(&5) does not work as you cannot take the address of a constant. This is not a failing of the F77 macros, but is inherent in the C language. For routines that are called in many places, it will be more convenient to write a wrap-up function in C that is to be called from the C main program and to put all of the F77 macros required into that function. This produces less efficient code, since there is an extra level of subroutine call. However, in many situations, the extra cost will be outweighed by the benefits of more transparent code.

The macro F77_CALL actually expands to the same thing as the macro F77_EXTERNAL_NAME, but is included as it is more descriptive of what is being done when calling a FORTRAN routine from C.

8.1 Thread Safety

Fortran code is not thread-safe, and therefore any C code that calls Fortran code will not be thread-safe unless extra work is done to make it so. The F77_LOCK macro is provided for this purpose. The argument to the macro is a block of code to be run. CNF defines a single global pthread mutex. The F77_LOCK macro firsts locks this mutex, then executes the code specified in its argument, then unlocks the mutex. If another thread already has the mutex locked, then the calling thread will block until the mutex is unlocked.

So any C code that may potentially need to be executed in a threaded context (for instance, C wrappers for Fortran subroutine libraries) should use the F77_LOCK macro to invoke each Fortran call:

```
F77_LOCK ( F77_CALL(silly2)( REAL_ARG(&a), REAL_ARG(&b),
                             INTEGER_ARG(&i), INTEGER_ARG(&j),
                             CHARACTER_ARG(fline), INTEGER_ARG(&fline_l),
                             LOGICAL_ARG(&x) TRAIL_ARG(fline) ); )
```

If this is done consistently, then it ensures that no two threads will attempt to run any Fortran code simultaneously.

9 More on Arrays

For most data types arrays are handled simply, using pointers as already demonstrated. However, for arrays of some types the data in the arrays must be converted back and forth between C and FORTRAN representations. Macros and functions are provided to facilitate the conversions.

Very often, the actual size of the FORTRAN array required will not be known until runtime so space for it must be allocated dynamically in a similar way to dynamic character strings.

Macros DECLARE_*type*_ARRAY_DYN and F77_CREATE_type_ARRAY are defined to do this. They are designed for 1-dimensional arrays, having just the name and the number of elements as parameters, but for Unix systems, at least, will work for multi-dimensional arrays.

For most types on all current systems, the CREATE_ARRAY macros will not actually allocate space as no conversion of data is necessary, but they are provided for contingency and completeness.

9.1 CHARACTER and LOGICAL Arrays

There are two versions of the macros for creating dynamic CHARACTER and LOGICAL arrays: F77_CREATE_CHARACTER_ARRAY will create a 1-dimensional array with the given number of elements, and F77_CREATE_CHARACTER_ARRAY_M will create an array whose size is defined by an integer specifying the number of dimensions and an array of integers specifying each dimension. Similarly F77_CREATE_LOGICAL_ARRAY and F77_CREATE_LOGICAL_ARRAY_M

Consider the following example of a C program which calls a FORTRAN subroutine which returns a CHARACTER array produced by setting to blank every non-blank element of a given array for which the corresponding element of a given LOGICAL array is TRUE. A LOGICAL output array is produced with TRUE in the element corresponding with each element of the CHARACTER array which has been reset, and FALSE elsewhere.

Example 8 – Import and export of arrays..

```
#include <stdio.h>
#include "f77.h"
F77_SUBROUTINE(str_reset)(CHARACTER_ARRAY(in), LOGICAL_ARRAY(lin),
                          INTEGER(dim1), INTEGER(dim2),
                          CHARACTER_ARRAY(out), LOGICAL_ARRAY(lout)
                          TRAIL(in) TRAIL(out) );

void main(){
char inarr[3][2][4]={{"Yes","No "},{"   ","   "},{"No ","Yes"}};
int inarr_length=4;
char outarr[3][2][4];
int outarr_length=4;
int lin[3][2]={{1,0},{1,1},{0,1}};
int lout[3][2];
DECLARE_CHARACTER_ARRAY(fin,3,2][4);
DECLARE_CHARACTER_ARRAY_DYN(fout);
DECLARE_LOGICAL_ARRAY(flin,3][2);
DECLARE_LOGICAL_ARRAY_DYN(flout);
DECLARE_INTEGER(dim1);
DECLARE_INTEGER(dim2);
int ndims=2;
int dims[2]={3,2};
int i,j;

   F77_CREATE_CHARACTER_ARRAY_M(fout,3,ndims,dims);
   F77_CREATE_LOGICAL_ARRAY_M(flout,ndims,dims);

   (void) cnfExprta(
      (char *)inarr, inarr_length, (char *)fin, fin_length, ndims, dims );
   (void) cnfExpla( (int *)lin, (F77_LOGICAL_TYPE *)flin, ndims, dims );

   dim1 = dims[0];
   dim2 = dims[1];

   F77_CALL(str_reset)( CHARACTER_ARRAY_ARG(fin), LOGICAL_ARRAY_ARG(flin),
                     INTEGER_ARG(&dim1), INTEGER_ARG(&dim2),
                     CHARACTER_ARRAY_ARG(fout), LOGICAL_ARRAY_ARG(flout)
                     TRAIL_ARG(fin) TRAIL_ARG(fout) );

   (void) cnfImprta
           ( fout, fout_length, outarr[0][0], outarr_length, ndims, dims );
   (void) cnfImpla( (F77_LOGICAL_TYPE *)flout, (int *)lout, ndims, dims );

   F77_FREE_CHARACTER(fout);
   F77_FREE_LOGICAL(flout);

   printf("i j in  lin out lout\n");
   for (j=0;j<3;j++){
      for (i=0;i<2;i++){
         printf("%d %d %c  %s  %c  %s\n",
            i, j, lin[j][i]?'T':'F', inarr[j][i],
                  lout[j][i]?'T':'F', outarr[j][i] );
      }
   }
}


      SUBROUTINE STR_RESET( ARRAY, LIN, DIM1, DIM2, OUT, LOUT )
*  Purpose:
*     Reset elements of an array

*  Arguments:
*     ARRAY(2,3)=CHARACTER*(*) (Given)
*        The array to be altered
*     LIN(2,3)=LOGICAL (Given)
*        The given LOGICAL array
*     DIM1=INTEGER (Given)
*        The first dimension of the arrays
*     DIM2=INTEGER (Given)
*        The second dimension of the arrays
*     OUT(2,3)=CHARACTER*(*) (Returned)
*     LOUT(2,3)=LOGICAL (Returned)

      IMPLICIT NONE
      INTEGER I, J
      INTEGER DIM1, DIM2
      CHARACTER*(*) ARRAY(2,3)
      CHARACTER*(*) OUT(2,3)
      LOGICAL LIN(2,3)
      LOGICAL LOUT(2,3)

      DO 20, J = 1, 3
         DO 10, I = 1, 2
            IF( LIN(I,J) .AND. (ARRAY(I,J) .NE. ' ') )THEN
               OUT(I,J) = ' '
               LOUT(I,J) = .TRUE.
            ELSE
               OUT(I,J) = ARRAY(I,J)
               LOUT(I,J) = .FALSE.
            END IF
10       ENDDO
20    ENDDO
```

```
      END
```

As an example of how to write a C function to be called from FORTRAN with array arguments, the above subroutine could be re-written in C as follows:

```
#include "f77.h"

F77_SUBROUTINE(str_reset)(CHARACTER_ARRAY(in_f), LOGICAL_ARRAY(lin_f),
                          INTEGER(dim1), INTEGER(dim2),
                          CHARACTER_ARRAY(out_f), LOGICAL_ARRAY(lout_f)
                          TRAIL(in_f) TRAIL(out_f) )
{
GENPTR_CHARACTER_ARRAY(in_f)
GENPTR_LOGICAL_ARRAY(lin_f)
GENPTR_INTEGER(dim1)
GENPTR_INTEGER(dim2)
GENPTR_CHARACTER_ARRAY(out_f)
GENPTR_LOGICAL_ARRAY(lout_f)

int i, j, nels, cpt;
char *in_c, *out_c;
int *lin_c, *lout_c;
int ndims=2;
int dims[2];

   dims[0] = *dim1;
   dims[1] = *dim2;
   nels = *dim1 * *dim2;

   in_c = cnfCreat( nels*(in_f_length+1) );
   out_c = cnfCreat( nels*(out_f_length+1) );
   lin_c = (int *)malloc( nels*sizeof(int) );
   lout_c = (int *)malloc( nels*sizeof(int) );
   cnfImprta( in_f, in_f_length, in_c, in_f_length+1, ndims, dims );
   cnfImpla( lin_f, lin_c, ndims, dims );

   cpt = 0;
   for(i=0;i<nels;i++){
      if( *(lin_c+i) && strlen( in_c+cpt ) ) {
          strcpy(out_c+cpt,"");
          *(lout_c+i) = 1;
      } else {
          strcpy( out_c+cpt, in_c+cpt );
          *(lout_c+i) = 0;
      }
      cpt += in_f_length+1;
   }

   cnfExprta( out_c, out_f_length+1, out_f, out_f_length, ndims, dims );
   cnfExpla( lout_c, lout_f, ndims, dims );

   cnfFree( in_c );
   cnfFree( out_c );
```

```
      free( lin_c );
      free( lout_c );
   }
```

9.2 Arrays of `pointer to char`

In C, arrays of character strings are often held as arrays of `pointer to char`. This allows strings of varying length and not necessarily in contiguous memory. CNF functions `cnfImprtap` and `cnfExprtap` can be used to import/export arrays of pointer to char from/to FORTRAN `CHARACTER` arrays. The following example shows how to do this. The FORTRAN subroutine, PRARR, prints the given `CHARACTER` array and returns it set to blank strings. The C program prints the strings before and after the call to PRARR.

Example 9 – IMPORT/EXPORT with arrays of pointers to char.

```
rlsaxp_101% more temp.c
#include "f77.h"
F77_SUBROUTINE(prarr)(CHARACTER_ARRAY(arr) TRAIL(arr));
main() {
DECLARE_CHARACTER_ARRAY(arr,12,3);
char *ptr[3]={"ajc","hello there","TEXT"};
int dims[1]=3;
int i;

for (i=0;i<3;i++) printf("%d:%s:\n",i,ptr[i]);
cnfExprtap(ptr,arr[0],12,1,dims);
F77_CALL(prarr)(CHARACTER_ARRAY_ARG(arr) TRAIL_ARG(arr));
cnfImprtap(arr[0],12,ptr,1,1,dims);
for(i=0;i<3;i++) printf("%d:%s:\n",i,ptr[i]);
}


      SUBROUTINE PRARR( ARR )
      CHARACTER*(*) ARR(3)
      INTEGER I

      DO 10
         PRINT *, ':', ARR(I), ':'
         ARR( I ) = ' '
10    CONTINUE
      END
```

9.3 POINTER Arrays

An array of pointers would need to be converted back and forth between the C and FORTRAN representations to cope with the possibility that the length of a C pointer is not the same as the length of a FORTRAN `INTEGER`. This can be done by declaring a suitably-sized FORTRAN array and converting each element using either `cnfCptr` or `cnfFptr`, according to the direction of conversion.

For example, to call a FORTRAN subroutine which returns an array of three pointers to real, the C code would need to be something like:

```
F77_REAL_TYPE * pntr[3]
DECLARE_POINTER_ARRAY(fpntr,3)

F77_CALL(getptr)(POINTER_ARRAY_ARG(fpntr))
/* Import the pointers to C */
for (i=0;i<3;i++) pntr[i]=(F77_REAL_TYPE *)cnfCptr(fpntr[i]);
```

See also The IMPORT and EXPORT macros (Section 10).

10 The IMPORT and EXPORT Macros

We have already seen that character strings and `LOGICAL` and `POINTER` variables have to be converted between the different forms used by FORTRAN and C, and the idea of "importing" a FORTRAN value to a C value, and "exporting" a C value to a FORTRAN value has been introduced with the CNF routines..

Potentially all the other types could differ so macros `F77_IMPORT_`*type*, `F77_IMPORT_`*type*`_ARRAY`, `F77_EXPORT_`*type* and `F77_EXPORT_`*type*`_ARRAY` are defined to copy the data as required – they will use CNF routines where appropriate. An additional *type* of `LOCATOR` is allowed for the `IMPORT/EXPORT` macros to handle character strings used as HDS locators. There are also macros `F77_IMPORT_CHARACTER_ARRAY_P` and `F77_EXPORT_CHARACTER_ARRAY_P` to handle the CHARACTER conversion if the C array is an array of pointers to char.

The `IMPORT/EXPORT_ARRAY` macros have arguments giving pointers to the data and the number of elements to be converted. This is assumed to be sufficient for both single and multi-dimensional arrays.

These macros impose a slight overhead in that they require both the FORTRAN and C variables to be set up and some copying done, even when this is not strictly necessary. However, they do protect against possible future problems and ease the problem of deciding whether and how the import/export should be done.

In the case of arrays, only pointers are copied unless a conversion really is required (as in the case of `CHARACTER` and `LOGICAL` arrays, for example).

A complication arises where the actual argument for a FORTRAN subroutine to be called from C is an array which is only returned. In that case, no exporting is required but the FORTRAN array must still be associated with the C array so that the FORTRAN subroutine knows where to store the results. For those types which require genuine conversion, a pointer to the FORTRAN array will have been set when the space was allocated but for others the pointer must be set to point to the actual C array. Macros `F77_ASSOC_`*type*`_ARRAY` are defined to do this where necessary. They are complementary to the `F77_CREATE_`*type*`_ARRAY` macros so you can include both to ensure that the pointer to the FORTRAN array is set correctly.

After use, the memory holding the FORTRAN array should be returned using an `F77_FREE_`*type* macro (which will do nothing if the `CREATE` macros for the type do not allocate space).

So, a C wrapper for the FORTRAN routine `str_reset` in the section on Handling `CHARACTER` and `LOGICAL` arrays could be written as follows:

Example 10 – Use of `IMPORT/EXPORT` macros.

```
#include "f77.h"
F77_SUBROUTINE(str_reset)( CHARACTER_ARRAY(array),
                           LOGICAL_ARRAY(lin),
                           INTEGER(dim1),
                           INTEGER(dim2),
                           CHARACTER_ARRAY(out),
                           LOGICAL_ARRAY(lout)
                           TRAIL(array)
                           TRAIL(out) );

void strReset( char *array,
               int array_length,
               int *lin,
               int dim1,
               int dim2,
               char *out,
               int out_length,
               int *lout ) {

DECLARE_CHARACTER_ARRAY_DYN(farray);
DECLARE_LOGICAL_ARRAY_DYN(flin);
DECLARE_INTEGER(fdim1);
DECLARE_INTEGER(fdim2);
DECLARE_CHARACTER_ARRAY_DYN(fout);
DECLARE_LOGICAL_ARRAY_DYN(flout);
int nels;

/* The dimensions of the arrays are being lied about */
/* calculate the number of elements */
   nels = dim1 * dim2;

/* Set up "given" arguments */
   F77_CREATE_CHARACTER_ARRAY( farray, array_length-1, nels );
   F77_EXPORT_CHARACTER_ARRAY(array, array_length, farray, farray_length, nels);
   F77_CREATE_LOGICAL_ARRAY( flin, nels );
   F77_EXPORT_LOGICAL_ARRAY( lin, flin, nels );
   F77_EXPORT_INTEGER( dim1, fdim1 );
   F77_EXPORT_INTEGER( dim2, fdim2 );
/* Set up "returned" arguments */
   F77_CREATE_CHARACTER_ARRAY( fout, out_length-1, nels );
   F77_ASSOC_CHARACTER_ARRAY( fout, out );
   F77_CREATE_LOGICAL_ARRAY( flout, nels );
   F77_ASSOC_LOGICAL_ARRAY( flout, lout );

   F77_CALL(str_reset)( CHARACTER_ARRAY_ARG(farray),
                        LOGICAL_ARRAY_ARG(flin),
                        INTEGER_ARG(&fdim1),
                        INTEGER_ARG(&fdim2),
                        CHARACTER_ARRAY_ARG(fout),
                        LOGICAL_ARRAY_ARG(flout)
                        TRAIL_ARG(farray)
                        TRAIL_ARG(fout) );

   F77_FREE_CHARACTER( farray );
   F77_FREE_LOGICAL( flin );
   F77_IMPORT_CHARACTER_ARRAY( fout, fout_length, out, out_length, nels );
   F77_FREE_CHARACTER( fout );
   F77_IMPORT_LOGICAL_ARRAY( flout, lout, nels );
   F77_FREE_LOGICAL( flout );

   return;
}
```

and the corresponding main routine would be:

```
#include <stdio.h>
#include "f77.h"
void strReset( char *array,
               int array_length,
               int *lin,
               int dim1,
               int dim2,
               char *out,
               int out_length,
               int *lout );

void main(){
char inarr[3][2][4]={{"Yes","No "},{"   ","   "},{"No ","Yes"}};
int inarr_length=4;
char outarr[3][2][4];
int outarr_length=4;
int lin[3][2]={{1,0},{1,1},{0,1}};
int lout[3][2];
int i,j;

   strReset(&inarr[0][0][0], 4, &lin[0][0], 3, 2,
            &outarr[0][0][0], 4, &lout[0][0] );

   printf("i j in  lin out lout\n");
   for (j=0;j<3;j++){
      for (i=0;i<2;i++){
         printf("%d %d %c  %s  %c  %s\n",
            i, j, lin[j][i]?'T':'F', inarr[j][i],
                  lout[j][i]?'T':'F', outarr[j][i] );
     }
   }
}
```

11 Subroutines and Functions as Arguments

Macros are provided to handle subroutine and function names passed as arguments. They correspond closely to the macros for handling normal data type arguments. The following example shows how to pass the name of an INTEGER function from a C program to a FORTRAN subroutine.

Example 11 – Passing names from C to FORTRAN.

A C program which calls a FORTRAN subroutine which needs the name of an INTEGER function as an argument.

```
#include "f77.h"

extern F77_SUBROUTINE(tst_ifun)( INTEGER_FUNCTION(name),
                                 INTEGER(status) );

extern F77_INTEGER_FUNCTION(ifun)();

main(){
DECLARE_INTEGER(status);

   status = 0;

   F77_CALL(tst_ifun)( INTEGER_FUNCTION_ARG(ifun),
                       INTEGER_ARG(&status) );

   printf( "Status set is: %d\n", status );

}
```

The FORTRAN subroutine:

```
*+ TST_IFUN - Call an integer function
      SUBROUTINE TST_IFUN( NAME, STATUS )

      INTEGER NAME
      EXTERNAL NAME
      INTEGER STATUS

      INTEGER I

      STATUS = NAME( STATUS )

      END
```

The INTEGER function:

```
*+ IFUN - A very simple FORTRAN INTEGER FUNCTION
      INTEGER FUNCTION IFUN( STATUS )

      INTEGER STATUS

      IFUN = STATUS + 99

      END
```

Corresponding macros are defined for other types of function and for FORTRAN subroutines.

Now suppose in the above example the subroutine TST_IFUN was written in C to be called from FORTRAN. The code would be something like:

Example 12 – Passing names from FORTRAN to C.

```
*+ TESTIFUN - Call a SUBROUTINE which requires a function name argument.
      PROGRAM TSTIFUN

      EXTERNAL IFUN
      INTEGER STATUS

      CALL TST_IFUN( IFUN, STATUS )
      PRINT *, 'STATUS is: ', STATUS

      END
```

```
#include "f77.h"

F77_INTEGER_FUNCTION(ifun)();

F77_SUBROUTINE(tst_ifun)(INTEGER_FUNCTION(name), INTEGER(status) ){

GENPTR_INTEGER_FUNCTION(name)
GENPTR_INTEGER(status)

*status = F77_EXTERNAL_NAME(name)( INTEGER_ARG(status) );
```

12 Other Approaches to Mixed Language Programming

The F77 macros and CNF functions described in this document provide a complete way of writing portable programs in a mixture of FORTRAN and C. All of the work necessary to provide the correct interface goes into writing the C routines. It is relatively painless to call C from FORTRAN, since the work of writing the interface need only be done once, but it can be annoying to have to write a lot of extra code every time that a FORTRAN routine is called from a C one. As mentioned in More on Calling FORTRAN from C (Section 8), it may be appropriate to write wrap-around routines when calling FORTRAN from C.

Another package that tackles the problem of mixing C and FORTRAN is one called CFORTRAN, written by Burkhard Burow of the University of Toronto. This will be available as part of the CERN library and could be provided on Starlink if required. This package allows you to write an interface layer between a user's code and a subroutine package such that neither side need be aware that the other is written in a foreign language. This is a crucial difference from the F77 macros, where the C code is written in the full knowledge that the function is being called from, or is to call, a FORTRAN routine. It is certainly possible to write a package that can be called either from FORTRAN or C using the F77 macros, but this does not occur automatically.

When using CFORTRAN, an extra level of subroutine call is always involved over what is strictly necessary using the macros described in this document. This results in less efficient code. However, when this is not a serious problem, there may be situations in which it is more appropriate to use the CFORTRAN system in preference to F77.

13 Compiling and Linking

Unless they are passing pointers to subprograms, FORTRAN programs do not need to be compiled in any special way when employing mixed language programming since they are not aware that the subprogram that they are calling is not written in FORTRAN. However, when pointers are passed using the mechanism described in Accessing Dynamic Memory from C and FORTRAN (Section 7.3), the FORTRAN code must include the statement:

```
INCLUDE 'CNF_PAR'
```

to define the function `CNF_PVAL`.

Type:

```
% cnf_dev
```

to define the link, `CNF_PAR`, to the required include file.

When compiling a C function that is to be called from FORTRAN, it should contain the line:

```
#include "f77.h"
```

to define the F77 macros and CNF functions[2].

On a Unix system, you can usually tell the C compiler where to look for header files with the `-I` qualifier to the cc command, *e.g.*:

```
% cc -I/star/include -c func.c
```

All FORTRAN `INCLUDE` and C header files for Starlink software are stored in the directory `/star/include`, and the object files for all Starlink libraries reside in `/star/lib`.

To link a FORTRAN program `prog.f` and a C function `sub.c` with the CNF library, first compile the C function and then compile and link the FORTRAN program:

```
% cc -c -I/star/include sub.c
% f77 prog.f sub.o -L/star/lib `cnf_link` -o prog
```

To link a C program `prog.c` with a FORTRAN subroutine `sub.f`, the procedure varies depending upon the system being used. It is usually best to try to do the link with the `f77` command as the correct FORTRAN libraries will then be searched. However, in some cases there is confusion over the main routine and either `cc` or `ld` must be used specifying all the required libraries.

For example, on Alpha/OSF1 it might be:

```
% f77 -c sub.f
% cc prog.c sub.o -L/star/lib `cnf_link` -lfor -lots -o prog
```

[2]The two CNF header files, `cnf.h` and `f77.h` are now identical. For legacy reasons it is acceptable to `#include` either or both in the code – just `f77.h` is preferred.

For ADAM tasks (see SG/4), much of the complication is removed by the task linking scripts alink and ilink which will accept a mixture of FORTRAN and C modules to compile. For example:

```
% alink task.f subr.c ...
```

or

```
% alink task.c subr.f ...
```

The CNF library will be linked automatically and `/star/include` searched for any required C header files.

The first program module specified for the ADAM link script must be the main routine of the ADAM task, which is written as a FORTRAN subroutine or C function with a single `INTEGER`, or `int *`, argument (see SUN/144 for details).

A Implementation Specific Details

As indicated several times earlier, many of the details of mixed language programming are implementation dependent. This section will deal in turn with each type of hardware that Starlink possesses. Given that programs can be written in a portable way, you may wonder if you need to know about the implementation specific details at all. This is in fact necessary when debugging programs, since the debugger will be working on the output of any macros that hide the implementation specific details from the programmer.

There is some duplication between the following subsections, one for each type of operating system, particularly in the examples. This has been done so that each section can be read separately from any other.

A.1 Sun

A.1.1 General

A Sun computer is based on a 32 bit architecture. Data can be addressed in multiples of 1, 2, 4, 8 or 16 bytes, a byte being 8 bits. References to FORTRAN and C in this subsection refer to the Sun FORTRAN and ANSI C compilers.

A.1.2 Data Types

There is a simple correspondence between Sun FORTRAN and C numeric variable types. The standard types are given in the upper part of

Table 1 and non-standard extensions in the lower part. These should generally be avoided for reasons of portability, however, they are provided since HDS (see SUN/92) has corresponding data types.

Although C defines unsigned data types of `unsigned char` (range 0 to 255), `unsigned short` (range 0 to 32767) and `unsigned int` (range 0 to $2^{32} - 1$), there are no corresponding unsigned data types in FORTRAN. There is also a C type called `long int`, however on Suns, this is the same as an `int`.

The C language does not specify whether variables of type `char` should be stored as signed or unsigned values. On Suns, they are stored as signed values in the range -128 to 127.

Similarly there is no C data type that corresponds to the FORTRAN data type of `COMPLEX`. However, since Sun FORTRAN passes all numeric variable by reference, a `COMPLEX` variable could be passed to a C subprogram where it might be handled as a structure consisting of two variables of type `float`.

A Sun FORTRAN `LOGICAL` value can be passed to a C `int`. Sun FORTRAN and C both use zero to represent a false value and anything else to represent a true value, so there is no problem with converting the data values.

A.1.3 External Names

The Sun FORTRAN compiler appends an underscore character to all external names that it generates. This applies to the names of subroutines, functions, labelled common blocks and block data subprograms.

type	Sun FORTRAN	Sun C
INTEGER	INTEGER	int
REAL	REAL	float
DOUBLE	DOUBLE PRECISION	double
LOGICAL	LOGICAL	int
	CHARACTER*1	char
CHARACTER	CHARACTER*n	char[n]
BYTE	BYTE	signed char
WORD	INTEGER*2	short int
UBYTE		unsigned char
UWORD		unsigned short int
POINTER	INTEGER	unsigned int

Table 1: Corresponding data types for Sun Solaris

A.1.4 Arguments

To understand how to pass arguments between Sun FORTRAN and C programs, it is necessary to understand the possible methods that the operating system can use for passing arguments and how each language makes use of them. There are three ways that an actual argument may be passed to a subroutine. What is actually passed as an argument should always be a four byte word. It is the interpretation of that word that is where the differences arise.

Sun FORTRAN passes all data types other than `CHARACTER` by reference, *i.e.* the address of the variable or array is put in the argument list. `CHARACTER` variables are passed by a mixture of reference and value. The argument list contains the address of the character variable being passed, but there is also an extra argument added at the end of the argument list for each character variable. This gives the actual length of the FORTRAN `CHARACTER` variable and so this datum is being passed by value. These extra arguments are hidden from the FORTRAN programmer, but must be explicitly included in any C routines.

C uses call by value to pass all variables, constants (except string constants), expressions, array elements, structures and unions that are actual arguments of functions. It uses call by reference to pass whole arrays, string constants and functions. C never uses call by descriptor as a default.

To pass a C variable of type `double` by value requires the use of two longwords in the argument list. Similarly, if a C structure is passed by value, then the number of bytes that it takes up in the argument list can be large. This is a dangerous practice and all structures should be passed by reference. Since, by default, Sun FORTRAN does not pass variables by value anyway, this should not give rise to any problems.

In Sun FORTRAN, the default argument passing mechanism can be overridden by use of the `%VAL` and `%REF` functions. These functions are not portable and should be avoided whenever possible. The `%DESCR` function provided in VAX FORTRAN is *not* provided on a Sun. In C there is no similar way of "cheating" as there is in FORTRAN; however, this is not necessary as the language allows more flexibility itself. For example, if you wish to pass a variable named `x` by reference rather than by value, you simply put `&x` as the actual argument instead of `x`.

Since C provides more flexibility in the mechanism of passing arguments than does FORTRAN, it is C that ought to shoulder the burden of handling the different mechanisms. All numeric variables and constants, array elements, whole arrays and function names should be passed into and out of C functions by reference. Numeric expressions will be passed from FORTRAN to C by reference and so the corresponding dummy argument in the C function should be declared to be of type "pointer to type". When C has a constant or an expression as an actual argument in a function call, it can only pass it by value. Sun FORTRAN cannot cope with this and so in a C program, all expressions should be assigned to variables before being passed to a FORTRAN routine.

Here are some examples to illustrate these points.

Example 13 – Passing arguments from Sun FORTRAN to C.

FORTRAN program:

```
      PROGRAM FORT1
      INTEGER A
      REAL B
      A = 1
      B = 2.0
      CALL C1( A, B )
      END
```

C function:

```
void c1_( int *a, float *b)
{
  int x;
  float y;

  x = *a;    /* x is now equal to 1 */
  y = *b;    /* y is now equal to 2.0 */

  printf( "x = %d\n", x );
  printf( "y = %f\n", y );
}
```

The C function name requires the underscore as the FORTRAN compiler generates this automatically.

In this first example, a Sun FORTRAN program passes an `INTEGER` and `REAL` variable to a C function. The values of these arguments are then assigned to two local variables. They could just as well have been used directly in the function by referring to the variables `*a` and `*b` instead of assigning their values to the local variables `x` and `y`. Since the FORTRAN program passes the actual arguments by reference, the dummy arguments used in the declaration of the C function should be a pointer to the variable that is being passed.

Now an example of calling a Sun FORTRAN subroutine from C.

Example 14 – Passing arguments from C to Sun FORTRAN.

C main program:

```
main()
{
 int  i = 2;             /* Declare i and initialize it.  */
 void fort2_( int *i );  /* Declare function fort2_. */

 fort2_( &i );           /* Call fort2.  */
}
```

FORTRAN subroutine:

```
      SUBROUTINE FORT2( I )
      INTEGER I

      PRINT *,I

      END
```

The C main function declares and initializes a variable, `i`, and declares a function `fort2_` (note the underscore). It calls `fort2_`, passing the address of the variable `i` rather than its value, as this is what the FORTRAN subroutine will be expecting.

As we have seen, the case of scalar numeric arguments is fairly straightforward, however, the passing of character variables between Sun FORTRAN and C is more complicated. Sun FORTRAN passes character variables by passing the address of the character variable and then adding an extra value to the argument list that is the size of the character variable. Furthermore, there is the point that FORTRAN deals with fixed-length, blank-padded strings, whereas C deals with variable-length, null-terminated strings. The simplest possible example of a character argument is given here as an illustration. Don't worry if it looks complicated, the F77 macros described in Section 5 hide all of these details from the programmer, and in a portable manner as well!

Example 15 – Passing character arguments from Sun FORTRAN to C.

FORTRAN program:

```
      PROGRAM FORT3
      CHARACTER STR*20

      CALL C3( STR )
      PRINT *,STR

      END
```

C function:

```
#include <stdio.h>                        /* Standard I/O functions */

void c3_( char *fortchar, int length )
```

```
{
  int  i;                              /* A loop counter */
  char  *string = "This is a string";  /* A string to be printed */

/* Copy the string to the function argument */
  strncpy( fortchar, string, length );

/* Pad the character argument with trailing blanks */
  for( i = strlen( string ) ; i < length ; i++ )
     fortchar[i] = ' ';
}
```

The second variable declaration in the C subprogram declares a local variable to be a string and initializes it. This string is then copied to the storage area that the subprogram argument points to, taking care not to copy more characters than the argument has room for. Finally any remaining space in the argument is filled with blanks, the null character being overwritten. You should always fill any trailing space with blanks in this way.

A.1.5 Function Values

The way that the return value of a function is handled is very much like a simple assignment statement. The value is actually returned in one or two of the registers of the CPU, depending on the size of the data type. Consequently there is no problem in handling the value of any function that returns a numerical value as long as the storage used by the value being returned and the value expected correspond (see Table 1 on page 40).

The case of a function that returns a character string is more complex. The way that Sun FORTRAN returns a character variable as a function value is to add two hidden extra entries to the beginning of the argument list. These are a pointer to a character variable and the value of the length of this variable. If a C function wishes to emulate a FORTRAN `CHARACTER` function, then you must explicitly add these two extra arguments to the C function. Any value that the C function returns will be ignored. Here is an example to illustrate this.

Example 16 – Use of a Sun FORTRAN character function.

FORTRAN program:

```
      PROGRAM CFUNC
      CHARACTER*(10) VAR, FUNC

      VAR = FUNC( 6 )
      PRINT *, VAR

      END
```

C function:

```
void func_( char *retval, int length, int *n )
{
  char *cp;
  int i, max;
/* Find the number of characters to be copied.  */
  if( *n < length )
     max = *n;
  else
     max = length;

/* Set a local character pointer equal to the return address.  */
  cp = retval;

/* Copy some asterisks to the "return value".  */
  for( i = 0 ; i < max ; i++ )
     *cp++ = '*';

/* Fill the rest of the string with blanks.  */
  for( ; i < length ; i++ )
     *cp++ = ' ';
}
```

The C function copies some asterisks into the location that Sun FORTRAN will interpret as the return value of the FORTRAN `CHARACTER` function. The number of such asterisks is specified by the single argument of the FORTRAN function and the rest of the string is filled with blanks.

A.1.6 Global Data

Although FORTRAN and C use different method for representing global data, it is actually very easy to mix them. If a Sun FORTRAN common block contains a single variable or array, then the corresponding C variable simply needs to be declared as `extern` and the two variables will use the same storage.

Example 17 – A labelled Sun FORTRAN common block containing a single variable.

FORTRAN common block:

```
      CHARACTER*(10) STRING
      COMMON /BLOCK/ STRING
```

C external variable:

```
extern char block_[10];
```

Note that the name of the C variable corresponds to the name of the FORTRAN common block, not the name of the FORTRAN variable. This example shows that you can use the same storage area for both Sun FORTRAN and C strings, however, you must still beware of the different way in which FORTRAN and C handle the end of a string.

If the FORTRAN common block contains more than one variable or array, then the C variables must be contained in a structure.

If you wish to access the Sun FORTRAN blank common block, then the corresponding C structure should be called `_BLNK__`.

Example 18 – A labelled Sun FORTRAN common block containing several variables.

FORTRAN common block:

```
            INTEGER I,J,K
            COMMON /NUMS/ I,J,K
```

C external variable:

```
      extern struct { int i,j,k; } nums_;
```

A.2 DEC Unix

A.2.1 General

This section applies for Alpha OSF/1, Ultrix/RISC and possibly other DEC Unix systems.

The machine specific details relating to mixed language programming are almost identical to those for the Sun and so the previous subsection should be consulted for more details. This is not to say that there are no differences between the DECstation and Sun compilers, merely that they do not generally impinge on the question of mixed language programming.

A.2.2 LOGICAL Values

One place where the DEC system may differ from the Sun is in how logical values are handled. The original FORTRAN compiler for the DECstation (FORTRAN for RISC) used the Sun interpretation of logical values, *i.e.* zero is false, non-zero is true. The more recent DEC FORTRAN compiler uses the VMS convention that only checks the lowest bit of a value, so 0 is false, 1 is true, 2 is false, 3 is true, *etc*. When DEC FORTRAN sets a `LOGICAL` variable to `TRUE`, all the bits in the data are set to 1, resulting in a numerical equivalent value of -1. Unfortunately this means that the correct value of the macros `F77_ISFALSE` and `F77_ISTRUE` used in a C function, depend on which FORTRAN compiler you are using. It is not possible to handle this automatically, so you must be sure to use the right values for the macros. The default assumption is that you are using the newer DEC FORTRAN compiler. Fortunately this is unlikely to be a problem in practice, since a `TRUE` value will normally be 1 or -1, and these values will be handled correctly by either compiler.

A.2.3 POINTERS on Alphas

The DEC Alpha machines can use addresses up to 64 bits long, but where FORTRAN `INTEGER`s are used to hold an address, only 32 bits can be held. However, the linker has flags -T and -D which can be used to ensure that allocated memory addresses will fit into 32 bits. The user generally does not have to worry about these, as they are inserted automatically if the relevant Starlink library link script (*e.g.* `hds_link`) is used.

A.3 VAX/VMS

A.3.1 General

A VAX computer is based on a 32 bit architecture. Data can be addressed as bytes (8 bits), words (16 bits), longwords (32 bits), quadwords (64 bits) or octawords (128 bits). The terminology is a hangover from the PDP-11 series of computers and the basic unit of storage on a VAX is the longword. References to FORTRAN and C in this subsection refer to the VAX FORTRAN and VAX C compilers produced by DEC.

A.3.2 Data Types

There is a simple correspondence between VAX FORTRAN and VAX C numeric variable types. The standard types are given in the upper part of

Table 2 and non-standard extensions in the lower part. These should generally be avoided for reasons of portability. However, they are provided since HDS (see SUN/92) has corresponding data types.

type	VAX FORTRAN	VAX C
INTEGER	INTEGER	int
REAL	REAL	float
DOUBLE	DOUBLE PRECISION	double
LOGICAL	LOGICAL	int
	CHARACTER*1	char
CHARACTER	CHARACTER*n	char[n]
BYTE	BYTE	char
WORD	INTEGER*2	short int
UBYTE		unsigned char
UWORD		unsigned short int
POINTER	INTEGER	unsigned int

Table 2: Corresponding data types for VAX/VMS

Although VAX C defines unsigned data types of `unsigned char` (range 0 to 255), `unsigned short` (range 0 to 32767) and `unsigned int` (range 0 to $2^{32} - 1$), there are no corresponding unsigned data types in FORTRAN. There is also a C type called `long int`; however in VAX C, this is the same as an `int`.

The C language does not specify whether variables of type `char` should be stored as signed or unsigned values. On VMS, they are stored as signed values in the range -128 to 127.

Similarly there is no C data type that corresponds to the FORTRAN data type of `COMPLEX`. However, since VAX FORTRAN passes all numeric variable by reference, a `COMPLEX` variable could be passed to a VAX C subprogram where it might be handled as a structure consisting of two variables of type `float`.

A VAX FORTRAN `LOGICAL` value can be passed to a VAX C `int`, but care must be taken over the interpretation of the value since VAX FORTRAN only considers the lower bit of the longword to be significant (0 is false, 1 is true) whereas VAX C treats any numerical value other than 0 as true. When VAX FORTRAN sets a logical value to true, it sets all the bits. This corresponds to a numerical value of minus one.

A.3.3 Arguments

To understand how to pass arguments between VAX FORTRAN and VAX C programs, it is necessary to understand the possible methods that VMS can use for passing arguments and how each language makes use of them. VMS defines a procedure calling standard that is used by all compilers written by DEC for the VMS operating system. This is described in the "Introduction to the VMS Run-Time Library" manual with additional information in the "Introduction to VMS System Services" manual. If you have a third party compiler that does not conform to this standard then you will not be able to mix the object code that it produces with that from DEC compilers. There are three ways that an actual argument may be passed to a subroutine. What is actually passed as an argument should always be a longword. It is the interpretation of that longword that is where the differences arise. Note the word *should* in the last but one sentence. VAX C will occasionally generate an argument that is longer than one longword. This is a violation of the VAX procedure calling standard. It causes no problems for pure VAX C programs, but is a potential source of problems for mixed language programs.

VAX FORTRAN passes all data types other than `CHARACTER` by reference, *i.e.* the address of the variable or array is put in the argument list. `CHARACTER` variables are passed by descriptor. The descriptor contains the type and class of descriptor, the length of the string and the address where the characters are actually stored.

VAX C uses call by value to pass all variables, constants (except string constants), expressions, array elements, structures and unions that are actual arguments of functions. It uses call by reference to pass whole arrays, string constants and functions. VAX C never uses call by descriptor as a default method of passing arguments.

To pass a VAX C variable of type `double` by value requires the use of two longwords in the argument list and so is a violation of the VAX procedure calling standard. The passing of a VAX C structure that is bigger that one longword is a similar violation. It is always better to pass C structures by reference, although this should not be a problem in practice since in the case of a pure VAX C program, everything is handled consistently and in the case of a mixture of FORTRAN and C, you would not normally pass variables by value anyway.

In VAX FORTRAN, the default argument passing mechanism can be overridden by use of the `%VAL`, `%REF` and `%DESCR` functions. These functions are not portable and should be avoided whenever possible. The only exception is that `%VAL` is used in Starlink software for passing pointer variables. In VAX C there is no similar way of "cheating" as there is in VAX FORTRAN; however, this is not necessary as the language allows more flexibility itself. For example, if you wish to pass a variable named `x` by reference rather than by value, you simply put `&x` as the actual argument instead of `x`. To pass something by descriptor, you need to construct the appropriate structure and pass the address of that. See the DEC manual "Guide to VAX C" for further details.

Since C provides more flexibility in the mechanism of passing arguments than does FORTRAN, it is C that ought to shoulder the burden of handling the different mechanisms. All numeric variables and constants, array elements, whole arrays and function names should be passed into and out of C functions by reference. Numeric expressions will be passed from VAX FORTRAN to VAX C by reference and so the corresponding dummy argument in the C function should be declared to be of type "pointer to type". When C has a constant or an expression as an actual argument in a function call, it can only pass it by value. VAX FORTRAN cannot cope with this and so in a VAX C program, all expressions should be assigned to variables before being passed to a FORTRAN routine.

Here are some examples to illustrate these points.

Example 19 – Passing arguments from VAX FORTRAN to VAX C.

FORTRAN program:

```
      PROGRAM FORT1
      INTEGER A
      REAL B
      A = 1
      B = 2.0
      CALL C1( A, B )
      END
```

C function:

```
void c1( int *a, float *b )
{
  int x;
  float y;

  x = *a;    /* x is now equal to 1 */
  y = *b;    /* y is now equal to 2.0 */

  printf( "x = %d\n", x );
  printf( "y = %f\n", y );
}
```

In this first example, a FORTRAN program passes an `INTEGER` and `REAL` variable to a C function. The values of these arguments are then assigned to two local variables. They could just as well have been used directly in the function by referring to the variables `*a` and `*b` instead of assigning their values to the local variables `x` and `y`. Since the VAX FORTRAN program passes the actual arguments by reference, the dummy arguments used in the declaration of the VAX C function should be a pointer to the variable that is being passed.

Now an example of calling a VAX FORTRAN subroutine from VAX C.

Example 20 – Passing arguments from VAX C to VAX FORTRAN.

C main program:

```
main()
{
 int  i = 2;            /* Declare i and initialize it.  */
 void fort2( int *i );  /* Declare function fort2. */

 fort2( &i );           /* Call fort2.  */
}
```

FORTRAN subroutine:

```
      SUBROUTINE FORT2( I )
      INTEGER I

      PRINT *,I

      END
```

The VAX C main function declares and initializes a variable, `i`, and declares a function `fort2`. It calls `fort2`, passing the address of the variable `i` rather than its value, as this is what the VAX FORTRAN subroutine will be expecting.

As we have seen, the case of scalar numeric arguments is fairly straightforward. However, the passing of `CHARACTER` variables between VAX FORTRAN and VAX C is more complicated. VAX FORTRAN passes `CHARACTER` variables by descriptor and VAX C must handle these descriptors. Furthermore, there is the point that FORTRAN deals with fixed-length, blank-padded strings, whereas C deals with variable-length, null-terminated strings. It is also worth noting that VAX/VMS machines handle `CHARACTER` arguments in a manner which is different from the usual Unix way. The simplest possible example of a `CHARACTER` argument is given here in all of its gory detail. You will be pleased to discover that this example is purely for illustration. The important point is that it is different from the Sun example and, anyway, the F77 macros described in Section 5 hide all of these differences from the programmer, thereby making the code portable.

Example 21 – Passing character arguments from VAX FORTRAN to VAX C.

FORTRAN program:

```
      PROGRAM FORT3
      CHARACTER STR*20

      CALL C3( STR )
      PRINT *,STR

      END
```

C function:

```
#include <descrip.h>                    /* VMS Descriptors */
#include <stdio.h>                      /* Standard I/O functions */

void c3( struct dsc$descriptor_s  *fortchar )
{
  int  i;                              /* A loop counter */
  char  *string = "This is a string";  /* A string to be printed */

/* Copy the string to the function argument */
  strncpy( fortchar->dsc$a_pointer, string, fortchar->dsc$w_length );

/* Pad the character argument with trailing blanks */
  for( i = strlen( string ) ; i < fortchar->dsc$w_length ; i++ )
     fortchar->dsc$a_pointer[i] = ' ';
}
```

The second variable declaration in the C subprogram declares a local variable to be a string and initializes it. This string is then copied to the storage area that the subprogram argument points to, taking care not to copy more characters than the argument has room for. Finally any remaining space in the argument is filled with blanks, the null character being overwritten. You should always fill any trailing space with blanks in this way. What should definitely not be done is to modify the descriptor to indicate the number of non blank characters that it now holds. The VAX FORTRAN compiler will not expect this to happen and it is likely to cause run-time errors. See the DEC manual "Guide to VAX C" for more details of handling descriptors in VAX C.

If an actual argument in a VAX FORTRAN routine is an array of characters, rather than just a single character variable, the descriptor that describes the data is different. It is defined by the macro `dsc$descriptor_a` instead of `dsc$descriptor_s`. This contains extra information about the number of dimensions and their bounds; however, this can generally be ignored since the first part of the `dsc$descriptor_a` descriptor is the same as the `dsc$descriptor_s` descriptor. This extra information can be unpacked from the descriptor, however, to do so would lead to non-portable code. It is generally better to use the address of the array that is passed in the descriptor and to pass any array dimensions as separate arguments. The C subroutine then has all of the information that it requires and can handle the data as an array or by using pointers, as the programmer sees fit. See example 4 for an illustration of this.

A.3.4 Function Values

The way that the return value of a function is handled is very much like a simple assignment statement. In practice, the value is actually returned in one or two of the registers of the CPU, depending on the size of the data type. Consequently there is no problem in handling the value of any function that returns a numerical value as long as the storage used by the value being returned and the value expected correspond (see Table 2 on page 46). If a VAX C function is treated as a `LOGICAL` function by VAX FORTRAN, there is no problem as long as the VAX C function ensures that it returns a value that will be interpreted correctly. The best thing to do is to make sure that the C function can only return zero (for false) or minus one (for true).

The case of a function that returns a character string is more complex. The way that VAX FORTRAN returns a `CHARACTER` variable as a function value is to add a hidden extra entry to the beginning of the argument list. This is a pointer to a character descriptor. If a VAX C function wishes to return a function value that VAX FORTRAN will interpret as a character string, then you must explicitly add an extra argument to the VAX C function and build the appropriate structure in your C function. This may seem rather complicated, but what it boils down to is that the following two segments of VAX FORTRAN are equivalent (but only in VAX FORTRAN).

Example 22 – Equivalence of a VMS character function and a VMS subroutine.

```
      CHARACTER*(10) RETURN
      CALL CHARFN( RETURN, A, B )
```

or

```
      CHARACTER*(10) RETURN, CHARFN
      RETURN = CHARFN( A, B )
```

If written as a function, CHARFN returns a value of type `CHARACTER`. It is left as an exercise for the reader to demonstrate that the above assertion is true using just FORTRAN.

A.3.5 Global Data

Although FORTRAN and C use different methods for representing global data, it is actually very easy to mix them. If a VAX FORTRAN common block contains a single variable or array, then the corresponding VAX C variable simply needs to be declared as `extern` and the two variables will use the same storage.

Example 23 – A VAX FORTRAN labelled common block containing a single variable

FORTRAN common block:

```
      CHARACTER*(10) STRING
      COMMON /BLOCK/ STRING
```

C external variable:

```
   extern char block[10];
```

Note that the name of the C variable corresponds to the name of the FORTRAN common block, not the name of the FORTRAN variable. This example shows that you can use the same storage area for both VAX FORTRAN and VAX C strings. However, you must still beware of the different way in which FORTRAN and C handle the end of a string.

If the FORTRAN common block contains more than one variable or array, then the C variables must be contained in a structure.

If you wish to access the VAX FORTRAN blank common block, then the corresponding VAX C structure should be called $BLANK.

Example 24 – A VAX FORTRAN labelled common block containing several variables.

FORTRAN common block:

```
      INTEGER I,J,K
      COMMON /NUMS/ I,J,K
```

C external variable:

```
   extern struct { int i,j,k; } nums;
```

A.4 Other Operating Systems

The F77 macros have been designed to cope with other systems as far as is possible. It should be possible to modify the include file `f77.h` to cope with most computers. The places where this may prove difficult, or even impossible, are likely to be due to arguments being passed in an unforeseen way.

The include file also declares the functions used for handling character strings. The declarations are written as function prototypes and assume that the C compiler will handle this feature of ANSI C. If a particular C compiler does not support this feature, then the header file could easily be modified to take this into account.

B Rationale for Mixed Language Programming

Starlink has historically been a "FORTRAN only" project. There are several reasons for this. Primarily it is because scientists have been brought up with FORTRAN and for most purposes it is perfectly adequate for our needs. However, there are some tasks for which FORTRAN is not really suitable. In such situations it may be better to write programs in a language other than FORTRAN, rather than try to persuade FORTRAN to do something that it is not suited to. Writing recursive procedures is the classic example, but there are many more. Starlink has recognised the need for a language other than FORTRAN by providing C compilers at all Starlink nodes.

There are in fact good reasons to avoid diversifying into trendy new languages unless it is absolutely necessary. Any substantial piece of software will require someone to support it long after the original author has moved on to other things and it is not reasonable to expect that person to have expertise in a large number of programming languages. However, for some purposes, FORTRAN 77 is simply not adequate. In fact some major parts of Starlink software have been written in other languages because of this. HDS. (See SUN/92.) is written in C (it was originally written in Bliss, in the days when even C was impractical because of restrictions in the early compilers). In the future, FORTRAN 90 will overcome many of the limitations that FORTRAN 77 has but, until that becomes readily available (and even after), some things are simply better written in C.

It is often the case that most of a program can be written in FORTRAN, leaving only a few tricky parts that cannot be written using standard (or even non standard) FORTRAN. An example of a task that cannot be performed using standard FORTRAN is getting some memory for use in your program. Admittedly, there are often system service subroutines available but these are virtually guaranteed to be non portable to other computers. Often a better approach is to write the tricky parts in C. This is exactly the approach that has been adopted for HDS. The problem then is how to pass data between FORTRAN routines and C functions. This document will describe how to do this. Clearly the details of passing information between program segments written in different languages will be machine dependent; however, there are also many important similarities. Despite any problems that may arise, it is easier to port programs written in a mixture of FORTRAN and C to other computer systems than to port programs written purely in FORTRAN that make use of machine-specific routines for system services.

How to mix FORTRAN and C in a way that is portable to all current Starlink hardware is described in

Sections 5 and 8.

It is quite likely that you will often want to use C to make use of something that the C run time library provides, such as allocating memory. This requirement is sufficiently common that a library of FORTRAN callable routines has already been provided to do exactly that. It is called PSX and is described in SUN/121. In many programs, use of the PSX library will remove the need to write any C code at all.

You may think that if you want to use C for part of a program then you should use C for all of the program. This may indeed be the best option; however, if you also want to call subroutines that are written in FORTRAN (*e.g.* just about any Starlink library), then you are going to be involved in mixed language programming anyway. The correct choice will depend on the circumstances.

Writing mixed language programs is not something that should be embarked upon lightly. There might be a better way of achieving the same result using just FORTRAN. The source code may not look as pretty, but if it runs effectively and efficiently then that is all that is required. If you can achieve what you want using standard FORTRAN then you should do so. If you cannot, then this document will tell you how to mix FORTRAN and C *in a portable way*. The programming language manuals of the computer manufacturers tell you how to mix languages on their own hardware, but achieving portability needs a little more thought.

Finally, if you are new to C, you should be aware that the way that things are normally done in C can be rather different from the way that they are normally done in FORTRAN. When I was new to C, I proudly showed someone one of my first C programs. “That’s not a C program,” they said, “That’s a FORTRAN program that’s written in C.” They were, of course, right. A useful book an C programming is Banahan [3]. This describes how to write programs in ANSI standard C and is written in an easy-going style. The author is not averse to criticizing C when he thinks that a feature of the language is not appropriate.

C Alphabetical List of F77 Macros

The list is alphabetical except that the generic *type* has highest priority. *type* may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

type
Declare a C function argument of the specified type

***type*_ARG**
Pass an argument of the specified type to a FORTRAN routine

***type*_ARRAY**
Declare a C function argument as an array of the specified type

***type*_ARRAY_ARG**
Pass an array argument of the specified type to a FORTRAN routine

***type*_FUNCTION**
Declare a C function argument as a FORTRAN-callable FUNCTION of the specified type

***type*_FUNCTION_ARG**
Pass a FORTRAN-callable FUNCTION of the specified type as an argument to a FORTRAN routine

CHARACTER_RETURN_ARG
Pass an argument that will be the return value of a CHARACTER FUNCTION

CHARACTER_RETURN_VALUE
Declare an argument that will be the return value of a CHARACTER FUNCTION

DECLARE_*type*
Declare a variable of the specified type

DECLARE_*type*_ARRAY
Declare an array of the specified type

DECLARE_*type*_ARRAY_DYN
Declare a dynamic array of the specified type

DECLARE_CHARACTER_DYN
Declare a dynamic FORTRAN CHARACTER variable

F77_*type*_FUNCTION
Declare a FORTRAN-callable function that returns a value of the specified type

F77_BLANK_COMMON
Refer to blank common

F77_BYTE_TYPE
Define the C type corresponding to the FORTRAN type BYTE

F77_CALL
Call a FORTRAN routine from C

F77_CHARACTER_ARG_TYPE
Define the type passed as a CHARACTER argument

F77_CHARACTER_ARRAY_ARG_TYPE
Define the type passed as a CHARACTER array argument

F77_CHARACTER_TYPE
Define the C type corresponding to the FORTRAN type CHARACTER

F77_CREATE_*type*_ARRAY
Create a dynamic FORTRAN array of type

F77_CREATE_CHARACTER
Create a dynamic FORTRAN CHARACTER variable

F77_CREATE_CHARACTER_ARRAY
Create a dynamic FORTRAN CHARACTER 1-D array

F77_CREATE_CHARACTER_ARRAY_M
Create a dynamic FORTRAN CHARACTER n-D array

F77_CREATE_LOGICAL_ARRAY_M
Create a dynamic FORTRAN LOGICAL n-D array

F77_DOUBLE_TYPE
Define the C type corresponding to the FORTRAN type DOUBLE PRECISION

F77_EXPORT_*type*
Export a C variable of the specified type to FORTRAN

F77_EXPORT_*type*_ARRAY
Export a C array of the specified type to FORTRAN

F77_EXPORT_CHARACTER_ARRAY_P
Export an array of pointers to char

F77_EXTERNAL_NAME
The external name of a function

F77_FALSE
The FORTRAN logical value FALSE

F77_FREE_*type*
Free a dynamic FORTRAN array or CHARACTER variable

F77_IMPORT_*type*
Import a FORTRAN variable of the specified type to C

F77_IMPORT_*type*_ARRAY
Import a FORTRAN array of the specified type to C

F77_IMPORT_CHARACTER_ARRAY_P
Import an array of pointers to char

F77_INTEGER_TYPE
Define the C type corresponding to the FORTRAN type INTEGER

F77_ISFALSE
Is this the FORTRAN logical value false?

F77_ISTRUE
Is this the FORTRAN logical value true?

F77_LOCK
: *Prevents code from being run simultaneously in two separate threads*

F77_LOGICAL_TYPE
: *Define the C type corresponding to the FORTRAN type LOGICAL*

F77_NAMED_COMMON
: *Refer to a named common block*

F77_REAL_TYPE
: *Define the C type corresponding to the FORTRAN type REAL*

F77_SUBROUTINE
: *Declare a FORTRAN-callable SUBROUTINE*

F77_TRUE
: *The FORTRAN logical value TRUE*

GENPTR_*type*
: *Generate a pointer to an argument of the specified type*

GENPTR_*type*_ARRAY
: *Generate a pointer to an array argument of the specified type*

GENPTR_*type*_FUNCTION
: *Generate a pointer to an argument which is a FORTRAN-callable FUNCTION of the specified type*

GENPTR_SUBROUTINE
: *Generate a pointer to an argument which is a FORTRAN-callable SUBROUTINE*

SUBROUTINE
: *Declare a C function argument as a FORTRAN-callable SUBROUTINE name*

SUBROUTINE_ARG
: *Pass a FORTRAN-callable SUBROUTINE name as an argument to a FORTRAN routine*

TRAIL
: *Declare hidden trailing arguments*

TRAIL_ARG
: *Pass the length of a CHARACTER argument to a FORTRAN routine*

D Classified List of F77 Macros

This appendix contains a list of the F77 macros, arranged by functionality.

D.1 Declaration of a C Function

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

F77_*type*_FUNCTION
: *Declare a FORTRAN-callable function that returns a value of the specified type*

F77_SUBROUTINE
: *Declare a FORTRAN-callable SUBROUTINE*

D.2 Arguments of a C Function

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

type

Declare a C function argument of the specified type

***type*_ARRAY**

Declare a C function argument as an array of the specified type

***type*_FUNCTION**

Declare a C function argument as a FORTRAN-callable FUNCTION of the specified type

CHARACTER_RETURN_VALUE

Declare an argument that will be the return value of a CHARACTER FUNCTION

SUBROUTINE

Declare a C function argument as a FORTRAN-callable SUBROUTINE name

TRAIL

Declare hidden trailing arguments

D.3 Generate Pointers to Arguments

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

GENPTR_*type*

Generate a pointer to an argument of the specified type

GENPTR_*type*_ARRAY

Generate a pointer to an array argument of the specified type

GENPTR_*type*_FUNCTION

Generate a pointer to an argument which is a FORTRAN-callable FUNCTION of the specified type

GENPTR_SUBROUTINE

Generate a pointer to an argument which is a FORTRAN-callable SUBROUTINE

D.4 Data Type Macros

F77_BYTE_TYPE

Define the C type corresponding to the FORTRAN type BYTE

F77_CHARACTER_TYPE

Define the C type corresponding to the FORTRAN type CHARACTER

F77_DOUBLE_TYPE

Define the C type corresponding to the FORTRAN type DOUBLE PRECISION

F77_INTEGER_TYPE

Define the C type corresponding to the FORTRAN type INTEGER

F77_LOGICAL_TYPE

Define the C type corresponding to the FORTRAN type LOGICAL

F77_POINTER_TYPE
: *Define the C type corresponding to the type POINTER*

F77_REAL_TYPE
: *Define the C type corresponding to the FORTRAN type REAL*

F77_UBYTE_TYPE
: *Define the C type corresponding to the type UBYTE*

F77_UWORD_TYPE
: *Define the C type corresponding to the type UWORD*

F77_WORD_TYPE
: *Define the C type corresponding to the type WORD*

D.5 Logical Value Macros

F77_FALSE
: *The FORTRAN logical value FALSE*

F77_ISFALSE
: *Is this the FORTRAN logical value false?*

F77_ISTRUE
: *Is this the FORTRAN logical value true?*

F77_TRUE
: *The FORTRAN logical value TRUE*

D.6 External Name Macro

F77_EXTERNAL_NAME
: *The external name of a function*

D.7 Common Block Macros

F77_BLANK_COMMON
: *Refer to blank common*

F77_NAMED_COMMON
: *Refer to a named common block*

D.8 Declaring Variables for Passing to a FORTRAN Routine

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

DECLARE_*type*
: *C declaration of a FORTRAN variable of the specified type*

DECLARE_*type*_ARRAY
: *C declaration of a FORTRAN array of the specified type*

DECLARE_*type*_ARRAY_DYN
: *C declaration of a dynamic FORTRAN array of the specified type*

F77_CREATE_*type*_ARRAY
: *Create a dynamic FORTRAN array of type*

DECLARE_CHARACTER_DYN
: *C declaration of a dynamic FORTRAN CHARACTER variable*

F77_CREATE_CHARACTER
: *Create a dynamic FORTRAN CHARACTER variable*

F77_CREATE_CHARACTER_ARRAY
: *Create a dynamic FORTRAN CHARACTER 1-D array*

F77_CREATE_CHARACTER_ARRAY_M
: *Create a dynamic FORTRAN CHARACTER n-D array*

F77_CREATE_LOGICAL_ARRAY_M
: *Create a dynamic FORTRAN LOGICAL n-D array*

F77_FREE_*type*
: *Free a dynamic FORTRAN array or CHARACTER variable*

D.9 Importing and Exporting Arguments

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

F77_EXPORT_*type*
: *Export a C variable to a FORTRAN variable of* `type`

F77_EXPORT_*type*_ARRAY
: *Export a C array to a FORTRAN array of* `type`

F77_EXPORT_CHARACTER_ARRAY_P
: *Export an array of pointers to char to a FORTRAN CHARACTER array*

F77_IMPORT_*type*
: *Import a FORTRAN variable of* `type` *to a C variable*

F77_IMPORT_*type*_ARRAY
: *Import a FORTRAN array of* `type` *to a C array*

F77_IMPORT_CHARACTER_ARRAY_P
: *Import a FORTRAN CHARACTER array to a C array of pointer to char*

F77_ASSOC_*type*_ARRAY
: *Associate a FORTRAN array of* `type` *with a C array*

D.10 Passing Arguments to a FORTRAN Routine

type may be one of: CHARACTER, DOUBLE, INTEGER, LOGICAL, REAL, BYTE, WORD, UBYTE, UWORD or POINTER.

***type*_ARG**
: *Pass an argument of the specified type to a FORTRAN routine*

***type*_ARRAY_ARG**
: *Pass an array argument of the specified type to a FORTRAN routine*

***type*_FUNCTION_ARG**
: *Pass a FORTRAN-callable FUNCTION of the specified type as an argument to a FORTRAN routine*

CHARACTER_RETURN_ARG
: *Pass an argument that will be the return value of a CHARACTER FUNCTION*

F77_CALL
: *Call a FORTRAN routine from C*

SUBROUTINE_ARG
: *Pass a FORTRAN-callable SUBROUTINE name as an argument to a FORTRAN routine*

TRAIL_ARG
: *Pass the length of a CHARACTER argument to a FORTRAN routine*

D.11 Thread Safety

F77_LOCK
: *Prevents code from being run simultaneously in two separate threads*

E Full Description of F77 Macros

This appendix contains a full description of each macro. It is in two sections: Generic Descriptions containing those macros which may be described generically for the various types and Specific Descriptions for those which need a specific description.

The effect of each macro is described and the expansion of the macro on each of the supported systems is given. The following classes are defined for the examples:

All systems	All supported systems
All Unix	All supported Unix systems

Not all the facilities listed here are available for VAX/VMS even if a VMS example is given. In some cases the macro expansions described here will not be correct for VAX/VMS. Consult the VMS Starlink documentation set for information on the VMS release.

N.B. It is important not to leave spaces around arguments in macros calls as these spaces are then included in the macro expansion on some systems, *i.e.* write `F77_SUBROUTINE(fred)`, not

`F77_SUBROUTINE( fred )`. This seems to be a bug in the offending compilers, but the problem is there none the less.

Many macros currently expand to an empty string on all currently supported systems. Nevertheless, the macros should still be used to guard against them being necessary on future systems.

E.1 Generic Descriptions

Unless otherwise stated, *type* is one of CHARACTER, INTEGER, REAL, DOUBLE, LOGICAL, BYTE, WORD, UBYTE, UWORD, LOCATOR or POINTER.

DECLARE_*type*
Declare a FORTRAN variable

Description:

Declare a variable that will be passed to a FORTRAN routine. This variable will be the actual argument of a call to a FORTRAN routine. (*type* not CHARACTER or LOCATOR.)

Invocation:

`DECLARE_type(arg)`

Arguments:

arg The variable being declared.

Examples:

`DECLARE_type(arg)`

will expand as follows:

All systems: `F77_type_TYPE arg`

where `F77_type_TYPE` expands to the appropriate C type.

Associated macro::

`DECLARE_CHARACTER`

DECLARE_*type*_ARRAY
Declare a FORTRAN array

Description:

Declare an array of the appropriate type that will be passed to a FORTRAN routine. This array will be the actual argument of a call to a FORTRAN routine. (*type* not CHARACTER or LOCATOR.)

Invocation:

```
DECLARE_type_ARRAY(arg,dims)
```

Arguments:

arg The array being declared.

dims
The dimensions of the array.

Examples:

```
DECLARE_type_ARRAY(arg,10)
```

will expand as follows:

All systems: `F77_type_TYPE arg[10]`

```
DECLARE_type_ARRAY(arg,2][3][4)
```

will expand as follows:

All systems: `F77_type_TYPE arg[2][3][4]`

where `F77_type_TYPE` expands to the appropriate C type.

Associated macro::

```
DECLARE_CHARACTER_ARRAY
```

DECLARE_*type*_ARRAY_DYN
Declare a dynamic *type* array

Description:

Declare a dynamic *type* array that will be passed to a FORTRAN routine using the *type*_ARRAY_ARG macro. Use this macro, in combination with the F77_CREATE_*type*_-ARRAY and F77_FREE_*type* macros, where the size of the array is not known until run time. (*type* not LOCATOR.)

Invocation:

```
DECLARE_type_ARRAY_DYN(arg)
```

Arguments:

arg The variable being declared.

Examples:

```
DECLARE_type_ARRAY_DYN(farg)
```

will expand as follows:

All systems: `F77_type_TYPE *farg`

```
DECLARE_CHARACTER_ARRAY_DYN(fstring)
```

will expand as follows:

All Unix:	`char *fstring; int fstring_length`
VAX/VMS:	`char *fstring; int fstring_length`
	`struct dsc$descriptor_a fstring_arg`

Notes:

On VMS, for CHARACTER, the expansion of the macro is quite complex. A pointer to a descriptor structure is declared in addition to a pointer to char (used to point to the actual string of characters) and an int variable to store the length of the array. The address of the descriptor is what is actually passed to the called FORTRAN routine.

F77_ASSOC_*type*_ARRAY
Associate a FORTRAN array with a C array.

Description:

For types which do not require separate memory allocated to hold the FORTRAN array, this macro ensures that the pointer to the FORTRAN array points to the memory allocated for the C array. (*type* not LOCATOR.)

Invocation:

```
F77_ASSOC_type_ARRAY(farg,carg)
```

Arguments:

farg

A pointer to the FORTRAN array

carg

A pointer to the C array

Examples:

```
F77_ASSOC_type_ARRAY(farg,carg)
```

type not CHARACTER, LOGICAL or POINTER will expand as follows:

All systems: `farg=carg`

type CHARACTER, LOGICAL or POINTER will expand as follows:

All systems:

F77_CREATE_*type*_ARRAY
Create an array of *type*.

Description:

These macros ensure that memory is available for arrays to be used as actual arguments for FORTRAN subroutines, assuming that space is already allocated for a corresponding C array. That is, they will only allocate additional memory for those types which require a non-null export or import. (*type* not CHARACTER or LOCATOR.)

Invocation:

`F77_CREATE_`*`type`*`_ARRAY(farg,nels)`

Arguments:

carg
A pointer to the C array

farg
A pointer to the FORTRAN array

nels
The number of elements required

Examples:

`F77_CREATE_`*`type`*`_ARRAY(farg,n)`

type LOGICAL will expand as follows:

All systems:
```
{int f77dims[1];f77dims[0]=n;
farg=cnfCrela(1,f77dims);}
```

type POINTER will expand as follows:

All systems:
```
farg=
(F77_POINTER_TYPE *)malloc(n*sizeof(F77_POINTER_TYPE))
```

All other *type*s (except CHARACTER) will expand as follows:

All systems:

Associated macros::

`F77_CREATE_CHARACTER_ARRAY`, `F77_ASSOC_`*`type`*`_ARRAY`, `F77_EXPORT_`*`type`*`_ARRAY`, `F77_IMPORT_`*`type`*`_ARRAY`

F77_EXPORT_*type*
Export a C variable to a FORTRAN variable.

Description:

Copies a C variable to a FORTRAN variable making any required changes to the data. (*type* not CHARACTER.)

Invocation:

```
F77_EXPORT_type(carg,farg)
```

Arguments:

carg
The C value

farg
The FORTRAN variable

Examples:

```
F77_EXPORT_type(carg,farg)
```

type LOGICAL will expand as follows:

All systems: `farg=carg?F77_TRUE:F77_FALSE`

type POINTER will expand as follows:

All systems: `farg=cnfFptr(carg)`

type LOCATOR will expand as follows:

All systems: `cnfExpch(carg,farg,DAT__SZLOC)`

All other *type*s will expand as follows:

All systems: `farg=carg`

Associated macro::

`F77_EXPORT_CHARACTER`

F77_EXPORT_*type*_ARRAY
Export an array of *type* from C to FORTRAN

Description:

Depending upon the type and system, the C array will be copied to the FORTRAN array, making any required changes to the data, or the pointer to the FORTRAN array will be set to point to the C array. (*type* not CHARACTER.)

Invocation:

```
F77_EXPORT_type_ARRAY(carg,farg,nels)
```

Arguments:

carg
A pointer to the C array

farg
A pointer to the FORTRAN array

nels
The number of elements to be exported

Examples:

```
F77_EXPORT_type_ARRAY(farg,carg,nels)
```

type LOGICAL will expand as follows:

All systems:
```
{int f77dims[1];f77dims[0]=nels;
cnfExpla(carg,farg,1,f77dims);}
```

type POINTER will expand as follows:

All systems:
```
{ int f77i; for(f77i=0;nels>f77i;f77i++) {
farg[f77i]=cnfFptr(carg[f77i]); }}
```

All other *type*s will expand as follows:

All systems:
```
farg=carg
```

Associated macro::

```
F77_EXPORT_CHARACTER_ARRAY
```

F77_FREE_*type*
Free a dynamic variable of *type*

Description:

Frees the space obtained by a previous F77_CREATE_*type*_ARRAY or F77_CREATE_-CHARACTER macro). Makes use of cnf functions where appropriate. If the associated CREATE macro was null, the FREE macro will be null.

Invocation:

`F77_FREE_type(arg)`

Arguments:

arg The variable as passed to the FORTRAN subroutine.

Examples:

```
F77_FREE_LOGICAL(flog)
```

will expand as follows:

All systems: `cnfFree((char *)flog)`

F77_IMPORT_*type*
Import a FORTRAN variable to a C variable.

Description:

Copies a FORTRAN variable to a C variable making any required changes to the data. (*type* not CHARACTER.)

Invocation:

```
F77_IMPORT_type(farg,carg)
```

Arguments:

farg

The C value

carg

The FORTRAN variable

Examples:

```
F77_IMPORT_type(farg,carg)
```

type LOGICAL will expand as follows:

All systems: `carg=F77_ISTRUE(farg)`

type POINTER will expand as follows:

All systems: `carg=cnfCptr(farg)`

type LOCATOR will expand as follows:

All systems: `cnfImpch(farg,DAT__SZLOC,carg)`

All other *type*s will expand as follows:

All systems: `carg=farg`

Associated macro::

```
F77_IMPORT_CHARACTER
```

F77_IMPORT_*type*_ARRAY
Import an array of *type* from FORTRAN to C

Description:

Depending upon the type and system, the FORTRAN array will be copied to the C array, making any required changes to the data, or the pointer to the C array will be set to point to the FORTRAN array. (*type* not CHARACTER.)

Invocation:

`F77_IMPORT_type_ARRAY(farg,carg,nels)`

Arguments:

farg
A pointer to the FORTRAN array

carg
A pointer to the C array

nels
The number of elements to be exported

Examples:

`F77_IMPORT_type_ARRAY(carg,farg,nels)`

type LOGICAL will expand as follows:

All systems:
```
{ int f77dims[1];f77dims[0]=nels;
cnfImpla(farg,carg,1,f77dims);}
```

type POINTER will expand as follows:

All systems:
```
{ int f77i;for(f77i=0;nels>f77i;f77i++){
carg[f77i]=cnfCptr(farg[f77i]); }}
```

All other *type*s will expand as follows:

All systems: `carg=farg`

Associated macro::

`F77_IMPORT_CHARACTER_ARRAY`

F77_*type*_FUNCTION
Declare a FORTRAN function

Description:

Declare a C function that will be called from FORTRAN as though it were a FORTRAN function of the appropriate type. (*type* not LOCATOR.)

Invocation:

```
F77_type_FUNCTION(name)
```

Arguments:

name

The name of the function to be declared.

Examples:

```
F77_type_FUNCTION(name)
```

will expand as follows:

All Unix:	`F77_type_TYPE name_`
VAX/VMS:	`F77_type_TYPE name`

where `F77_type_TYPE` expands to the appropriate C type.

GENPTR_*type*
Generate a pointer to an argument

Description:

Ensure that there exists a pointer of the appropriate type to the variable that has been passed as an actual argument from FORTRAN to a C routine. Since FORTRAN usually passes arguments by reference, the pointer is commonly available directly from the argument list, so this macro is null. (*type* not LOCATOR.)

Invocation:

```
GENPTR_type(arg)
```

Arguments:

arg The dummy argument.

Examples:

```
GENPTR_type(arg)
```

will expand as follows:

All systems:

Associated macro::

```
GENPTR_CHARACTER
```

GENPTR_*type*_ARRAY
Generate a pointer to an array argument

Description:

Ensure that there exists a pointer of the appropriate type to the array that has been passed as an actual argument to the C routine. Since FORTRAN usually passes arguments by reference, the pointer is commonly available directly from the argument list, so this macro is null. (*type* not LOCATOR.)

Invocation:

```
GENPTR_type_ARRAY(arg)
```

Arguments:

arg The dummy argument.

Examples:

```
GENPTR_type_ARRAY(arg)
```

will expand as follows:

All systems:

Associated macro::

```
GENPTR_CHARACTER_ARRAY
```

GENPTR_*type*_FUNCTION
Generate a pointer to a FUNCTION argument

Description:

Ensure that there exists a pointer of the appropriate type to the FORTRAN FUNCTION that has been passed as an actual argument from FORTRAN to a C routine. Since FORTRAN usually passes arguments by reference, the pointer is commonly available directly from the argument list, so this macro is null. (*type* not LOCATOR.)

Invocation:

```
GENPTR_type_FUNCTION(name)
```

Arguments:

name

The dummy argument.

Examples:

```
GENPTR_type_FUNCTION(name)
```

will expand as follows:

All systems:

Notes:

The dummy argument should have been declared with the *type*_FUNCTION macro.

type
Declare a *type* argument

Description:

Declare a C function argument, given that the actual argument will be a variable of the appropriate type, passed from a FORTRAN program. (*type* not LOCATOR.)

Invocation:

type`(arg)`

Arguments:

arg The dummy argument to be declared.

Examples:

type`(arg)`

will expand as follows:

All Unix:	`F77_`*type*`_TYPE *arg`
VAX/VMS:	`F77_`*type*`_TYPE *const arg`

where `F77_`*type*`_TYPE` expands to the appropriate C type.

Associated macro::

`CHARACTER`

*type*_ARG
Pass a *type* argument to a FORTRAN routine

Description:

Pass an argument of the appropriate type to a FORTRAN routine. The argument should be the address of the variable. (*type* not LOCATOR.)

Invocation:

```
type_ARG(p_arg)
```

Arguments:

p_arg

A pointer to the actual argument being passed.

Examples:

```
type_ARG(&arg)
```

will expand as follows:

All systems: `&arg`

Associated macro::

`CHARACTER_ARG`

*type*_ARRAY
Declare a array argument

Description:

Declare a C function argument, given that the actual argument will be an array of the appropriate type, passed from a FORTRAN program. (*type* not LOCATOR.)

Invocation:

`type_ARRAY(arg)`

Arguments:

arg The dummy argument to be declared.

Examples:

`type_ARRAY(arg)`

will expand as follows:

All Unix:	`F77_type_TYPE *arg`
VAX/VMS:	`F77_type_TYPE *const arg`

where `F77_type_TYPE` expands to the appropriate C type.

Associated macro::

`CHARACTER_ARRAY`

*type*_ARRAY_ARG
Pass an array argument to a FORTRAN routine

Description:

Pass an array argument of the appropriate type to a FORTRAN routine. The argument should be the address of the array. (*type* not LOCATOR.)

Invocation:

```
type_ARRAY_ARG(p_arg)
```

Arguments:

p_arg

A pointer to the actual array being passed.

Examples:

```
type_ARRAY_ARG(arg)
```

will expand as follows:

All systems: `(F77_type_TYPE *)arg`

Notes:

The cast in the expansion for Unix ensures that multi-dimensional arrays (arrays of arrays), for example as declared by DECLARE_*type*_ARRAY, may be passed.

Associated macro::

`CHARACTER_ARRAY_ARG`

*type*_FUNCTION
Declare a FUNCTION argument

Description:

Declare a C function argument, to be a FORTRAN-callable FUNCTION of the specified type, passed from a FORTRAN program. (*type* is one of CHARACTER, INTEGER, REAL, DOUBLE, LOGICAL, BYTE, WORD, UBYTE, UWORD or POINTER.)

Invocation:

```
type_FUNCTION(arg)
```

Arguments:

arg The dummy argument to be declared.

Examples:

```
type_FUNCTION(arg)
```

will expand as follows:

All Systems: `F77_type_TYPE (*F77_EXTERNAL_NAME(arg))()`

where `F77_type_TYPE` and `F77_EXTERNAL_NAME` expand appropriately for the platform.

*type*_FUNCTION_ARG
Pass a FUNCTION argument to a FORTRAN routine

Description:

Pass a FORTRAN-callable FUNCTION of the appropriate type to a FORTRAN routine. The argument should be the address of the function being passed. (*type* is one of CHARACTER, INTEGER, REAL, DOUBLE, LOGICAL, BYTE, WORD, UBYTE, UWORD or POINTER.)

Invocation:

```
type_FUNCTION_ARG(p_arg)
```

Arguments:

p_arg

A pointer to the FUNCTION being passed.

Examples:

```
type_FUNCTION_ARG(arg)
```

will expand as follows:

All systems: `F77_EXTERNAL_NAME(arg)`

where `F77_EXTERNAL_NAME` expands appropriately for the platform.

E.2 Specific Descriptions

CHARACTER
Declare a CHARACTER argument

Description:

Declare a C function argument, given that the actual argument will be a CHARACTER variable passed from a FORTRAN program.

Invocation:

```
CHARACTER(arg)
```

Arguments:

arg The dummy argument to be declared.

Examples:

```
CHARACTER(x)
```

will expand as follows:

All Unix:	`char *x`
VAX/VMS:	`struct dsc$descriptor_s *x_arg`

Notes:

On a VAX/VMS system, the macro expands to a pointer to a descriptor whereas on other systems it expands to a pointer to char.

CHARACTER_ARG
Pass a CHARACTER argument to a FORTRAN routine

Description:

Pass a CHARACTER argument to a FORTRAN routine. The argument should be the address of a CHARACTER variable.

Invocation:

```
CHARACTER_ARG(p_arg)
```

Arguments:

p_arg

A pointer to the actual argument being passed.

Examples:

```
CHARACTER_ARG(charg)
```

will expand as follows:

All Unix:	`charg`
VAX/VMS:	`charg_arg`

CHARACTER_ARRAY
Declare a CHARACTER array argument

Description:

Declare a C function argument, given that the actual argument will be a CHARACTER array passed from a FORTRAN program.

Invocation:

```
CHARACTER_ARRAY(arg)
```

Arguments:

arg The dummy argument to be declared.

Examples:

```
CHARACTER_ARRAY(x)
```

will expand as follows:

All Unix:	`char *x`
VAX/VMS:	`struct dsc$descriptor_a *x_arg`

CHARACTER_ARRAY_ARG
Pass a CHARACTER array argument to a FORTRAN routine

Description:

Pass a CHARACTER array argument to a FORTRAN routine. The argument should be the address of a CHARACTER array.

Invocation:

```
CHARACTER_ARRAY_ARG(p_arg)
```

Arguments:

p_arg

A pointer to the actual array being passed.

Examples:

```
CHARACTER_ARRAY_ARG(charg)
```

will expand as follows:

All Unix:	`(char *)charg`
VAX/VMS:	`charg_arg`

Notes:

The cast in the expansion for Unix ensures that multi-dimensional arrays (arrays of arrays), for example as declared by DECLARE_*type*_ARRAY, may be passed.

CHARACTER_RETURN_ARG
Pass argument(s) that will be the return value of a FORTRAN CHARACTER FUNCTION

Description:

Pass the function return value argument(s) to a FORTRAN CHARACTER FUNCTION. There is no corresponding dummy argument in the FORTRAN FUNCTION, but the compiler generates an extra argument specifying the address and possibly another one, specifying the length of the value to be returned. The argument should be the address of a FORTRAN CHARACTER variable.

Invocation:

```
CHARACTER_RETURN_ARG(arg)
```

Arguments:

arg The hidden dummy argument to be declared.

Examples:

```
CHARACTER_RETURN_ARG(x)
```

will expand as follows:

All Unix:	`x ,int x_length`
VAX/VMS:	`x_arg`

CHARACTER_RETURN_VALUE
Declare argument(s) that will be the return value of a FORTRAN CHARACTER FUNCTION

Description:

Declare the C function argument(s) to return the value of a FORTRAN CHARACTER FUNCTION. There is no corresponding actual argument in the FORTRAN call but the compiler generates an extra argument specifying the address and possibly another one, specifying the length of the value to be returned.

Invocation:

```
CHARACTER_RETURN_VALUE(arg)
```

Arguments:

arg The hidden dummy argument to be declared.

Examples:

```
CHARACTER_RETURN_VALUE(x)
```

will expand as follows:

All Unix:	`char *x ,int x_length`
VAX/VMS:	`struct dsc$descriptor_s *x_arg`

DECLARE_CHARACTER
Declare a CHARACTER variable

Description:

Declare a CHARACTER variable that will be passed to a FORTRAN routine. This variable will be the actual argument of a call to a FORTRAN routine.

Invocation:

```
DECLARE_CHARACTER(arg,length)
```

Arguments:

arg The variable being declared.

length
The length of the character string.

Examples:

```
DECLARE_CHARACTER(C,50)
```

will expand as follows:

All Unix: `char C[50]; const int C_length = 50`

VAX/VMS:
```
char C[50]; const int C_length = 50;
struct dsc$descriptor_s C_descr =
{50, DSC$K_DTYPE_T, DSC$K_CLASS_S, C };
struct dsc$descriptor_s *C_arg =&C_descr
```

Notes:

On VMS, the expansion of the macro is quite complex. A char array is declared as well as an int variable to store the length of the array. There is also a descriptor and a pointer to that descriptor. The address of the descriptor is what is actually passed to the called FORTRAN routine.

DECLARE_CHARACTER_ARRAY
Declare a CHARACTER array

Description:

Declare a CHARACTER array that will be passed to a FORTRAN routine. This array will be the actual argument of a call to a FORTRAN routine.

Invocation:

```
DECLARE_CHARACTER_ARRAY(arg,length,dims)
```

Arguments:

arg The array being declared.

length
The length of the character string.

dims
The dimensions of the array.

Examples:

```
DECLARE_CHARACTER_ARRAY(C,50,10)
```

will expand as follows:

All Unix:	`char C[10][50]; const int C_length`
VAX/VMS:	`char C[10][50]; const int C_length = 50;`
	`struct dsc$descriptor_s C_descr =`
	`{50, DSC$K_DTYPE_T, DSC$K_CLASS_S, C };`
	`struct dsc$descriptor_s *C_arg =&C_descr`

Notes:

On VMS, the expansion of the macro is quite complex. A char array is declared as well as an int variable to store the length of the array. There is also a descriptor and a pointer to that descriptor. The address of the descriptor is what is actually passed to the called FORTRAN routine.

DECLARE_CHARACTER_DYN
Declare a CHARACTER variable

Description:

Declare a CHARACTER variable that will be passed to a FORTRAN routine using the CHARACTER_ARG macro. Use this macro, in combination with the F77_CREATE_-CHARACTER and F77_FREE_CHARACTER macros, where the length of the CHARACTER string is not known until run time.

Invocation:

```
DECLARE_CHARACTER_DYN(arg)
```

Arguments:

arg The variable being declared.

Examples:

```
DECLARE_CHARACTER_DYN(fstring)
```

will expand as follows:

All Unix:	`char *fstring; int fstring_length`
VAX/VMS:	`char *fstring; int fstring_length`
	`struct dsc$descriptor_s *fstring_arg`

Notes:

On VMS, the expansion of the macro is quite complex. A pointer to a descriptor structure is declared in addition to a pointer to char (used to point to the actual string of characters) and an int variable to store the length of the array. The address of the descriptor is what is actually passed to the called FORTRAN routine.

F77_BLANK_COMMON
Refer to blank common

Description:

Expands to the external name of blank common on the computer in use. This is used in declaring an external structure in C that overlays the FORTRAN blank common block.

Invocation:

```
F77_BLANK_COMMON
```

Examples:

```
F77_BLANK_COMMON
```

will expand as follows:

All Unix	`_BLNK__`
VAX/VMS:	`$BLANK`

```
extern struct { int i,j,k;} F77_BLANK_COMMON;
```

declares an external structure to use the same storage as the FORTRAN blank common.

```
F77_BLANK_COMMON.i
```

refers to component i of the above structure.

F77_BYTE_TYPE
Define the type BYTE

Description:

Define the C type that corresponds to the FORTRAN type BYTE.

Invocation:

```
F77_BYTE_TYPE
```

Examples:

```
F77_BYTE_TYPE
```

will expand as follows:

All Unix:	`signed char`
VAX/VMS:	`char`

F77_CALL
Call a FORTRAN routine from C

Description:
Call a FORTRAN subroutine or function from a C routine.

Invocation:

```
F77_CALL(name)
```

Arguments:

name
The name of the FORTRAN routine being called.

Examples:

```
F77_CALL(suba)
```

will expand as follows:

All Unix:	`suba_`
VAX/VMS:	`suba`

Notes:

This macro is just a shorthand for F77_EXTERNAL_NAME. It is more expressive to use F77_CALL rather than F77_EXTERNAL_NAME when calling a routine.

F77_CHARACTER_ARG_TYPE
Define the type of a FORTRAN CHARACTER argument

Description:

Defines the C type that corresponds to the type of a FORTRAN CHARACTER argument.

Invocation:

```
F77_CHARACTER_ARG_TYPE
```

Examples:

```
F77_CHARACTER_ARG_TYPE
```

will expand as follows:

All Unix:	`char`
VAX/VMS:	`struct dsc$descriptor_s`

Notes:

The type of the CHARACTER argument passed to a FORTRAN subroutine is not the same as the CHARACTER_TYPE on VMS so this macro is provided. It is unlikely to be used directly.

F77_CHARACTER_ARRAY_ARG_TYPE
Define the type of a FORTRAN CHARACTER array argument

Description:

Defines the C type that corresponds to the type of a FORTRAN CHARACTER array argument.

Invocation:

```
F77_CHARACTER_ARRAY_ARG_TYPE
```

Examples:

```
F77_CHARACTER_ARRAY_ARG_TYPE
```

will expand as follows:

All Unix:	`char`
VAX/VMS:	`struct dsc$descriptor_a`

Notes:

The type of the CHARACTER array argument passed to a FORTRAN subroutine is not the same as the CHARACTER_TYPE on VMS so this macro is provided. It is unlikely to be used directly.

F77_CHARACTER_TYPE
Define the type CHARACTER

Description:

Define the C type that corresponds to the FORTRAN type CHARACTER.

Invocation:

```
F77_CHARACTER_TYPE
```

Examples:

```
F77_CHARACTER_TYPE
```

will expand as follows:

All systems: `char`

F77_CREATE_CHARACTER
Create a FORTRAN CHARACTER variable

Description:

Create a CHARACTER variable that will be passed to a FORTRAN routine using the CHARACTER_ARG macro. Use this macro, in combination with the DECLARE_CHARACTER_-DYN and F77_FREE_CHARACTER macros, where the length of the CHARACTER string is not known until run time. A pointer to the actual string of characters and an integer variable giving the length of the string are set.

Invocation:

```
F77_CREATE_CHARACTER(arg,length)
```

Arguments:

arg The variable being created.

length
The length of the character string. This will usually be a variable name or expression of type int.

Examples:

```
F77_CREATE_CHARACTER(fstring,strlen(cstring))
```

will expand as follows:

All Unix:	`fstring_length = strlen(cstring);`
	`fstring = cnfCref(fstring_length)`
VAX/VMS:	`fstring_arg = cnfCref(strlen(cstring));`
	`fstring = fstring_arg->pointer;`
	`fstring_length = fstring_arg->length`

Notes:

On VMS, the expansion of the macro is quite complex. A descriptor structure and a pointer to it are set up in addition to the pointer to the actual string of characters and the length of the string. (The address of the descriptor is what is actually passed to the called FORTRAN routine.)

F77_CREATE_CHARACTER_ARRAY
Create a FORTRAN CHARACTER array

Description:

Create a CHARACTER array that will be passed to a FORTRAN routine using the CHARACTER_ARRAY_ARG macro. Use this macro, in combination with the DECLARE_-CHARACTER_ARRAY_DYN and F77_FREE_CHARACTER macros, where the size of the CHARACTER array is not known until run time. A pointer to the actual string of characters and an integer variable giving the length of the string are set.

Invocation:

```
F77_CREATE_CHARACTER_ARRAY(arg,length,nels)
```

Arguments:

arg The variable being created.

length
The length of the character string. This will usually be a variable name or expression of type int.

nels
The number of elements.

Examples:

```
F77_CREATE_CHARACTER_ARRAY(fstring,strlen(cstring),nels)
```

will expand as follows:

All Unix:
```
{ int f77dims[1];f77dims[0]=nels;
fstring=cnfCrefa(strlen(cstring),1,f77dims);
fstring_length=strlen(cstring);}
```

VAX/VMS:
```
{ int f77dims[1];f77dims[0]=nels;
fstring_arg = cnfCrefa(strlen(cstring),1,f77dims);
fstring = fstring_arg->pointer;
fstring_length = fstring_arg->length;}
```

Notes:

On VMS, the expansion of the macro is quite complex. A descriptor structure and a pointer to it are set up in addition to the pointer to the actual array of strings and the length of the strings. (The address of the descriptor is what is actually passed to the called FORTRAN routine.)

F77_CREATE_CHARACTER_ARRAY_M
Create a FORTRAN CHARACTER array (n-D)

Description:

Create an n-D CHARACTER array that will be passed to a FORTRAN routine using the CHARACTER_ARRAY_ARG macro. Use this macro, in combination with the DECLARE_-CHARACTER_ARRAY_DYN and F77_FREE_CHARACTER macros, where the size of the CHARACTER array is not known until run time. A pointer to the actual string of characters and an integer variable giving the length of the string are set.

Invocation:

```
F77_CREATE_CHARACTER_ARRAY_M(arg,length,ndims,dims)
```

Arguments:

arg The variable being created.

length
The length of the character string. This will usually be a variable name or expression of type int.

ndims
The number of dimensions.

dims
A 1-D array holding the ndims dimensions.

Examples:

```
F77_CREATE_CHARACTER_ARRAY_M(fstring,strlen(cstring),ndims,dims)
```

will expand as follows:

All Unix:	`fstring=cnfCrefa(strlen(cstring),ndims,dims);`
	`fstring_length=strlen(cstring)`
VAX/VMS:	`fstring_arg = cnfCrefa(strlen(cstring),ndims,dims);`
	`fstring = fstring_arg->pointer;`
	`fstring_length = fstring_arg->length`

Notes:

On VMS, the expansion of the macro is quite complex. A descriptor structure and a pointer to it are set up in addition to the pointer to the actual array of strings and the length of the strings. (The address of the descriptor is what is actually passed to the called FORTRAN routine.)

F77_CREATE_LOGICAL_ARRAY_M
Create a FORTRAN LOGICAL array (n-D)

Description:

Create a LOGICAL array that will be passed to a FORTRAN routine using the LOGICAL_-ARRAY_ARG macro. Use this macro, in combination with the DECLARE_LOGICAL_-ARRAY_DYN and F77_FREE_LOGICAL macros, where the size of the LOGICAL array is not known until run time.

Invocation:

```
F77_CREATE_LOGICAL_ARRAY_M(arg,ndims,dims)
```

Arguments:

arg The array being created.

ndims
The number of dimensions.

dims
A 1-D array holding the ndims dimensions.

Examples:

```
F77_CREATE_LOGICAL_ARRAY_M(flog,ndims,dims)
```

will expand as follows:

All systems: `flog=cnfCrela(ndims,dims);`

F77_DOUBLE_TYPE
Define the type DOUBLE PRECISION

Description:

Define the C type that corresponds to the FORTRAN type DOUBLE PRECISION.

Invocation:

```
F77_DOUBLE_TYPE
```

Examples:

```
F77_DOUBLE_TYPE
```

will expand as follows:

All systems: `double`

F77_EXPORT_CHARACTER
Export a C variable to a FORTRAN variable.

Description:
Copies a C variable to a FORTRAN variable making any required changes to the data.

Invocation:

```
F77_EXPORT_CHARACTER(carg,farg,len)
```

Arguments:

carg
The C value

farg
The FORTRAN variable

len The length of the FORTRAN string

Examples:

```
F77_EXPORT_CHARACTER(carg,farg,len)
```

will expand as follows:

All systems: `cnfExprt(carg,farg,len)`

Associated macro::

```
F77_IMPORT_CHARACTER
```

F77_EXPORT_CHARACTER_ARRAY
Export a CHARACTER array from C to FORTRAN

Description:

The C array will be copied to the FORTRAN array, making any required changes to the data

Invocation:

```
F77_EXPORT_CHARACTER_ARRAY(carg,lc,farg,lf,nels)
```

Arguments:

carg
A pointer to the C array

lc The length of the C strings

farg
A pointer to the FORTRAN array

lf The length of the FORTRAN strings

nels
The number of elements to be exported

Examples:

```
F77_EXPORT_CHARACTER_ARRAY(farg,lf,carg,lc,nels)
```

will expand as follows:

```
{ int f77dims[1];f77dims[0]=nels;
cnfExprta(carg,lc,farg,lf,1,f77dims);}
```

Associated macro::

```
F77_IMPORT_CHARACTER_ARRAY
```

F77_EXPORT_CHARACTER_ARRAY_P Export an array of pointers to char from C to a FORTRAN CHARACTER array.

Description:

The strings pointed to by the specified number of elements of the C array will be copied to the FORTRAN array, making any required changes to the data.

Invocation:

```
F77_EXPORT_CHARACTER_ARRAY_P(carg,farg,lf,nels)
```

Arguments:

carg
A pointer to the C array

farg
A pointer to the FORTRAN array

lf The length of the FORTRAN strings

nels
The number of elements to be exported

Examples:

```
F77_EXPORT_CHARACTER_ARRAY_P(carg,farg,lc,nels)
```

will expand as follows:

All systems:
```
{ int f77dims[1];f77dims[0]=nels;
cnfExprtap(carg,farg,lf,1,f77dims);}
```

Associated macro::

```
F77_IMPORT_CHARACTER_ARRAY_P
```

F77_EXTERNAL_NAME
The external name of a function

Description:

Define the external name of a C function. This may have such things as trailing underscores.

Invocation:

```
F77_EXTERNAL_NAME
```

Examples:

```
F77_EXTERNAL_NAME(name)
```

will expand as follows:

All Unix:	`name_`
VAX/VMS:	`name`

F77_FALSE
The logical value FALSE

Description:

Expand to the number that FORTRAN treats as a logical value of FALSE.

Invocation:

```
F77_FALSE
```

Examples:

```
F77_FALSE
```

will expand as follows:

All systems: `0`

Notes:

FORTRAN and C might not interpret the same numerical value as the same logical value.

F77_IMPORT_CHARACTER
Import a FORTRAN variable to a C variable.

Description:

Copies a FORTRAN CHARACTER string to a C string, making any necessary changes to the data

Invocation:

```
F77_IMPORT_CHARACTER(farg,len,carg)
```

Arguments:

farg
The FORTRAN string

len The length of the FORTRAN string

carg
The C string

Examples:

```
F77_IMPORT_CHARACTER(farg,len,carg)
```

will expand as follows:

All systems: `cnfImprt(farg,len,carg)`

Associated macro::

```
F77_EXPORT_CHARACTER
```

F77_IMPORT_CHARACTER_ARRAY
Import a CHARACTER array from FORTRAN to C.

Description:

The FORTRAN array will be copied to the the C array, making any required changes to the data.

Invocation:

```
F77_IMPORT_CHARACTER_ARRAY(farg,len_f,carg,len_c,nels)
```

Arguments:

farg
A pointer to the FORTRAN array

len_f
The length of each element of the FORTRAN array

carg
A pointer to the C array

len_c
The length of each element of the C array

nels
The number of elements to be exported

Examples:

```
F77_IMPORT_CHARACTER_ARRAY(farg,lf,carg,lc,nels)
```

will expand as follows:

All systems:
```
{ int f77dims[1];f77dims[0]=nels;
cnfImprta(farg,lf,carg,lc,1,f77dims);}
```

Associated macro::

```
F77_EXPORT_CHARACTER_ARRAY
```

F77_IMPORT_CHARACTER_ARRAY_P
Import a FORTRAN CHARACTER array to a C array of pointers to char.

Description:

The FORTRAN array will be copied to the series of C strings pointed at by the elements of the C array of pointers. If there is room (determined by the given maximum string length) strings will be null-terminated. Any required changes to the data will be made.

Invocation:

```
F77_IMPORT_CHARACTER_ARRAY_P(farg,len_f,carg,len_c,nels)
```

Arguments:

farg
A pointer to the FORTRAN array

len_f
The length of each element of the FORTRAN array

carg
A pointer to the C array

len_c
The maximum length of the C strings, including terminating null if required.

nels
The number of elements to be exported

Examples:

```
F77_IMPORT_CHARACTER_ARRAY_P(farg,lf,carg,lc,nels)
```

will expand as follows:

All systems:
```
{ int f77dims[1];f77dims[0]=nels;
cnfImprtap(farg,lf,carg,lc,f77dims);}
```

Associated macro::

```
F77_EXPORT_CHARACTER_ARRAY_P
```

F77_INTEGER_TYPE
Define the type INTEGER

Description:

Define the C type that corresponds to the FORTRAN type INTEGER.

Invocation:

```
F77_INTEGER_TYPE
```

Examples:

```
F77_INTEGER_TYPE
```

will expand as follows:

All systems: `int`

F77_ISFALSE
Is this the FORTRAN logical value false?

Description:

Does the argument of the macro evaluate to a value that FORTRAN would treat as a LOGICAL false?

Invocation:

```
if( F77_ISFALSE(var) ) ...
```

Arguments:

var	The name of the value to be tested.

Examples:

```
F77_ISFALSE(var)
```

will expand as follows:

Solaris:	`( ! ( var ) )`
OSF/1:	`( ! ( (var)&1 ) )`
VAX/VMS:	`( ! ( (var)&1 ) )`

Notes:

- The VAX FORTRAN and DEC FORTRAN for RISC compilers only use the lowest bit for the logical flag. Hence 0 = false, 1 = true, 2 = false, 3 = true, etc.
- The Sun FORTRAN compiler uses zero = false, non zero = true.
- The FORTRAN for RISC compiler (from MIPS) on the DECstation uses zero = false, non zero = true. This means that the correct value of this C macro depends on which FORTRAN compiler is being used.

F77_LOCK
Prevents code from being run simultaneously in two separate threads

Description:

Any C code that may need to be used in a threaded context should use this macro should to prevent Fortran code being run simultaneously in two separate threads, with consequent danger of unsynchronised memory access. The macro locks the global CNF mutex, then executes the code specified in the argument, and then unlocks the mutex. If the mutex is currently locked by another thread (e.g. due to the use of F77_LOCK in the other thread), then the calling thread blocks until the other thread releases the mutex.

Invocation:

```
F77_LOCK(code)
```

Arguments:

code

Any arbitrary C code. Typically, this will be an invocation of a Fortran subroutine.

Examples:

```
F77_LOCK( result = F77_CALL(sim)( nel, data, status ); )
```

Notes:

This macro invokes the cnfLock and cnfUnlock functions to lock and unlock the global mutex.

F77_ISTRUE
Is this the FORTRAN logical value true?

Description:

Does the argument of the macro evaluate to a value that FORTRAN would treat as a LOGICAL true?

Invocation:

```
if( F77_ISTRUE(var) ) ...
```

Arguments:

var The name of the value to be tested.

Examples:

```
F77_ISTRUE(var)
```

will expand as follows:

Solaris:	`( var )`
OSF/1:	`( (var)&1 )`
VAX/VMS:	`( (var)&1 )`

Notes:

- The VAX FORTRAN and DEC FORTRAN for RISC compilers only use the lowest bit for the logical flag. Hence 0 = false, 1 = true, 2 = false, 3 = true, etc.
- The Sun FORTRAN compiler uses zero = false, non zero = true.
- The FORTRAN for RISC compiler (from MIPS) on the DECstation uses zero = false, non zero = true. This means that the correct value of this C macro depends on which FORTRAN compiler is being used.

F77_LOGICAL_TYPE
Define the type LOGICAL

Description:

Define the C type that corresponds to the FORTRAN type LOGICAL.

Invocation:

```
F77_LOGICAL_TYPE
```

Examples:

```
F77_LOGICAL_TYPE
```

will expand as follows:

All systems: `int`

F77_NAMED_COMMON
Refer to a named common block

Description:

Expand to the external name of a named common block on the computer in use. This is used in declaring an external structure in C that overlays a FORTRAN named common block.

Invocation:

```
F77_NAMED_COMMON(name)
```

Arguments:

name

The name of the common block.

Examples:

```
F77_NAMED_COMMON(name)
```

will expand as follows:

All Unix:	`name_`
VAX/VMS:	`name`

```
extern struct {int i,j,k;} F77_NAMED_COMMON(block);
```

declares an external structure to use the same storage as the FORTRAN named common block.

```
F77_NAMED_COMMON(block).i
```

refers to component i of the above structure.

F77_POINTER_TYPE
Define the type POINTER

Description:

Define the C type that corresponds to a FORTRAN integer used as a pointer.

Invocation:

```
F77_POINTER_TYPE
```

Examples:

```
F77_POINTER_TYPE
```

will expand as follows:

All systems: `unsigned int`

F77_REAL_TYPE
Define the type REAL

Description:

Define the C type that corresponds to the FORTRAN type REAL.

Invocation:

```
F77_REAL_TYPE
```

Examples:

```
F77_REAL_TYPE
```

will expand as follows:

All systems: `float`

F77_SUBROUTINE
Declare a SUBROUTINE

Description:

Declare a C function that will be called from FORTRAN as though it were a subroutine.

Invocation:

```
F77_SUBROUTINE(name)
```

Arguments:

name

The name of the function to be declared.

Examples:

```
F77_SUBROUTINE(name)
```

will expand as follows:

All Unix:	`void name_`
VAX/VMS:	`void name`

F77_TRUE
The logical value TRUE

Description:

Expand to the number that FORTRAN treats as a logical value of TRUE.

Invocation:

```
F77_TRUE
```

Examples:

```
F77_TRUE
```

will expand as follows:

Solaris:	`1`
OSF/1:	`-1`
VAX/VMS:	`-1`

Notes:

FORTRAN and C might not interpret the same numerical value as the same logical value.

F77_UBYTE_TYPE
Define the type UBYTE

Description:

Define the C type that corresponds to the type UBYTE.

Invocation:

```
F77_UBYTE_TYPE
```

Examples:

```
F77_UBYTE_TYPE
```

will expand as follows:

All systems: `unsigned char`

F77_UWORD_TYPE
Define the type UWORD

Description:

Define the C type that corresponds to the type UWORD.

Invocation:

```
F77_UWORD_TYPE
```

Examples:

```
F77_UWORD_TYPE
```

will expand as follows:

All systems: `unsigned short int`

F77_WORD_TYPE
Define the type WORD

Description:

Define the C type that corresponds to the FORTRAN type WORD.

Invocation:

```
F77_WORD_TYPE
```

Examples:

```
F77_WORD_TYPE
```

will expand as follows:

All systems: `short int`

GENPTR_CHARACTER
Generate a pointer to a CHARACTER argument

Description:

Ensure that there exists a pointer to the character variable that has been passed as an actual argument to the C routine. Also generate a variable that contains the length of the actual character argument and call it 'arg_length'.

Invocation:

```
GENPTR_CHARACTER(arg)
```

Arguments:

arg The dummy argument.

Examples:

```
GENPTR_CHARACTER(x)
```

will expand as follows:

All Unix:

VAX/VMS:
```
char *x = x_arg->dsc$a_pointer;

int x_length = x_arg->dsc$w_length;
```

Notes:

On Unix systems, this macro is null, but on a VAX/VMS system, this macro actually declares the variables that are the pointer to the character string and the integer that contains its length.

GENPTR_CHARACTER_ARRAY
Generate a pointer to a CHARACTER array argument

Description:

Ensure that there exists a pointer to the character array that has been passed as an actual argument to the C routine. Also generate a variable that contains the length of the actual character argument and call it 'arg_length'.

Invocation:

```
GENPTR_CHARACTER_ARRAY(arg)
```

Arguments:

arg The dummy argument.

Examples:

```
GENPTR_CHARACTER_ARRAY(x)
```

will expand as follows:

All Unix:

VAX/VMS:
```
char *x = x_arg->dsc$a_pointer;

int x_length = x_arg->dsc$w_length;
```

Notes:

On Unix systems, this macro is null, but on a VAX/VMS system, this macro actually declares the variables that are the pointer to the character string and the integer that contains its length.

GENPTR_SUBROUTINE
Generate a pointer to a SUBROUTINE argument.

Description:

Ensure that there exists a pointer to the subroutine that has been passed as an actual argument from FORTRAN to the C function. Since FORTRAN usually passes arguments by reference, the pointer is commonly available directly from the argument list, so this macro is null.

Invocation:

```
GENPTR_SUBROUTINE(name)
```

Arguments:

name

The dummy argument.

Examples:

```
GENPTR_SUBROUTINE(name)
```

will expand as follows:

All systems:

Notes:

The dummy argument should have been declared with the SUBROUTINE macro.

SUBROUTINE
Declare a SUBROUTINE argument.

Description:

Declare a C function argument, given that the actual argument will be a SUBROUTINE name passed from a FORTRAN program.

Invocation:

```
SUBROUTINE(name)
```

Arguments:

name

The dummy argument to be declared.

Examples:

```
SUBROUTINE(name)
```

will expand as follows:

All systems: `void (*name)()`

SUBROUTINE_ARG
Pass a SUBROUTINE argument to a FORTRAN routine.

Description:

Pass a SUBROUTINE argument to a FORTRAN routine. The argument should be a pointer to a subroutine designed to be called from a FORTRAN program.

Invocation:

```
SUBROUTINE_ARG(p_name)
```

Arguments:

p_name

A pointer to the actual subroutine to be used.

Examples:

```
SUBROUTINE_ARG(name)
```

will expand as follows:

All systems: `name`

TRAIL
Declare hidden trailing arguments

Description:

Declare an argument on those machines that put an extra value at the end of the argument list to specify the length of a CHARACTER variable or array element.

Invocation:

```
TRAIL(arg)
```

Arguments:

arg	The name of the CHARACTER or CHARACTER array dummy argument.

Examples:

```
TRAIL(arg)
```

will expand as follows:

32bit Unix:	`,int arg_length`
Some 64bit Unix:	`,long arg_length`
VAX/VMS:	

TRAIL_ARG
Pass the length of a CHARACTER argument to a FORTRAN routine

Description:

Pass the length of a CHARACTER argument or a CHARACTER array argument element length to a FORTRAN routine if the FORTRAN routine expects to receive it as a separate argument. The corresponding integer variable is handled automatically where it is needed.

Invocation:

```
TRAIL_ARG(arg)
```

Arguments:

arg The name of the CHARACTER or CHARACTER array actual argument being passed.

Examples:

```
TRAIL_ARG(arg)
```

will expand as follows:

All Unix:	`,arg_length`
VAX/VMS:	

F Classified List of CNF Functions

F.1 Import a FORTRAN String to C

cnfCreib
: *Create a temporary C string and import a blank filled FORTRAN string into it*

cnfCreim
: *Create a temporary C string and import a FORTRAN string into it*

cnfImpb
: *Import a FORTRAN string into a C string, retaining trailing blanks*

cnfImpbn
: *Import no more than max characters from a FORTRAN string into a C string, retaining trailing blanks*

cnfImpch
: *Import a given number of characters from a FORTRAN string into an array of char*

cnfImpn
: *Import no more than max characters from a FORTRAN string into a C string*

cnfImprt
: *Import a FORTRAN string into a C string*

cnfImprta
: *Import a FORTRAN CHARACTER array into a C array*

cnfImprtap
: *Import a FORTRAN CHARACTER array into a C array of pointers to char*

F.2 Export a C String to FORTRAN

cnfExpch
: *Export a given number of characters from an array of char into a FORTRAN string*

cnfExpn
: *Export a C string to a FORTRAN string, copying given a maximum number of characters*

cnfExprt
: *Export a C string to a FORTRAN string*

cnfExprta
: *Export a C string array to a FORTRAN CHARACTER array*

cnfExprtap
: *Export a C array of pointers to char, to a FORTRAN CHARACTER array*

F.3 String Lengths

cnfLenc
: *Find the length of a C string*

cnfLenf
: *Find the length of a FORTRAN string*

F.4 Miscellaneous String Handling

cnfCopyf
: *Copy one FORTRAN string to another FORTRAN string*

cnfCreat
: *Create a temporary C string and return a pointer to it*

cnfCref
: *Create a temporary FORTRAN string and return a pointer to it*

cnfCrefa
: *Create a temporary FORTRAN CHARACTER array and return a pointer to it*

cnfFree
: *Free allocated space*

cnfFreef
: *Return temporary FORTRAN string space*

F.5 LOGICAL Array Handling

cnfCrela
: *Create a temporary FORTRAN LOGICAL array and return a pointer to it*

cnfImpla
: *Import a FORTRAN LOGICAL array into a C int array*

cnfExpla
: *Export a C int array into a FORTRAN LOGICAL array*

F.6 Memory and Pointer Handling

cnfCalloc
: *Allocate space that may be accessed from C and FORTRAN*

cnfCptr
: *Convert a FORTRAN pointer to a C pointer*

cnfFptr
: *Convert a C pointer to a FORTRAN pointer*

cnfFree
: *Free allocated space*

cnfMalloc
: *Allocate space that may be accessed from C and FORTRAN*

cnfRegp
: *Register a pointer for use from both C and FORTRAN*

cnfUregp
: *Unregister a pointer previously registered using cnfRegp*

CNF_PVAL
: *Expand a FORTRAN pointer to its full value (FORTRAN function)*

G CNF C Routine Descriptions

cnfCalloc
Allocate space that may be accessed from C and FORTRAN

Description:

This function allocates space in the same way as the standard C calloc() function, except that the pointer to the space allocated is automatically registered (using `cnfRegp`) for use from both C and FORTRAN. This means that the returned pointer may subsequently be converted into a FORTRAN pointer of type F77_POINTER_TYPE (using `cnfFptr`) and back into a C pointer (using `cnfCptr`). The contents of the space may therefore be accessed from both languages.

Invocation:

```
cpointer = cnfCalloc( nobj, size );
```

Arguments:

size_t nobj (Given)
The number of objects for which space is required.

size_t size (Given)
The size of each object.

Returned Value:

void ∗cnfCalloc
A registered pointer to the allocated space, or NULL if the space could not be allocated.

Notes:

- As with calloc(), the allocated space is initialised to zero bytes.
- The space should be freed using `cnfFree` when no longer required.

cnfCopyf
Copy one FORTRAN string to another FORTRAN string

Description:

The FORTRAN string in source_f is copied to dest_f. The destination string is filled with trailing blanks or truncated as necessary.

Invocation:

```
cnfCopyf( source_f, source_len, dest_f, dest_len )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

char ∗dest_f (Returned via pointer)
A pointer to the output FORTRAN string

int dest_len (Given)
The length of the output FORTRAN string

cnfCptr
Convert a FORTRAN pointer to a C pointer

Description:

Given a FORTRAN pointer, stored in a variable of type F77_POINTER_TYPE, this function returns the equivalent C pointer. Note that this conversion is only performed if the C pointer has originally been registered (using `cnfRegp`) for use from both C and FORTRAN. All pointers to space allocated by `cnfCalloc` and `cnfMalloc` are automatically registered in this way.

Invocation:

```
cpointer = cnfCptr( fpointer )
```

Arguments:

F77_POINTER_TYPE fpointer (Given)
The FORTRAN pointer value.

Returned Value:

void $*$cnfCptr
The equivalent C pointer.

Notes:

- A NULL value will be returned if the C pointer has not previously been registered for use from both C and FORTRAN, or if the FORTRAN pointer value supplied is zero.

cnfCreat
Create a temporary C string and return a pointer to it

Description:

Create a temporary C string and return a pointer to it. The space allocated to the C string is 'length' characters and is initialized to the null string.

Invocation:

```
pointer = cnfCreat( length )
```

Arguments:

int length (Given)

The length of the space to be allocated in characters.

Returned Value:

char ∗cnfCreat

A pointer to the storage that has been allocated by this routine.

Notes:

- If the argument is given as N then there is room to store N-1 characters plus a trailing null character in a C string.
- If the routine could not create the space, then it returns a null pointer.

cnfCref
Create a temporary FORTRAN CHARACTER string and return a pointer to it.

Description:

Memory is obtained for a FORTRAN CHARACTER string of the specified length and a pointer is returned which may be passed from C to a FORTRAN subroutine. The string is not initialised to blanks.

Invocation:

```
string_f = cnfCref( string_f_len )
```

Arguments:

int string_f_len (Given)

The required length of the FORTRAN string.

Returned Value:

F77_CHARACTER_ARG_TYPE ∗cnfCref

A pointer to the storage that has been allocated by this routine. Note that this is not necessarily the location of the string of characters.

Notes:

- If the routine could not create the space, then it returns a null pointer.
- This function will usually be called via the F77 F77_CREATE_CHARACTER macro which will also provide a pointer to the actual string of characters.

cnfCrefa
Create a temporary FORTRAN CHARACTER array and return a pointer to it.

Description:

Memory is obtained for a FORTRAN CHARACTER array, of the specified dimensions and a pointer is returned which may be passed from C to a FORTRAN subroutine. The array is not initialised to blanks.

Invocation:

```
string_f = cnfCrefa( string_f_len, ndims, dims )
```

Arguments:

int string_f_len (Given)
The maximum length of the FORTRAN string elements of the array.

int ndims (Given)
The number of dimensions of the FORTRAN array

const int ∗dims (Given)
A 1-D array giving the dimensions of the FORTRAN array.

Returned Value:

F77_CHARACTER_ARRAY_ARG_TYPE ∗cnfCrefa
A pointer to the storage that has been allocated by this routine. Note that this is not necessarily the location of the strings of characters.

Notes:

- If the routine could not create the space, then it returns a null pointer.
- This function will usually be called via the F77 F77_CREATE_CHARACTER_ARRAY macro which will also provide a pointer to the actual strings of characters.

cnfCreib
Create a temporary C string and import a FORTRAN string into it including trailing blanks

Description:

Create a temporary C string, import a FORTRAN string into it, retaining trailing blanks and return a pointer to this C string. The length of the C string that is created is just long enough to hold the FORTRAN string (including any trailing blanks), plus the null terminator.

Invocation:

```
pointer = cnfCreib( source_f, source_len )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

Returned Value:

char ∗cnfCreib
A pointer to the temporary storage location

Notes:

If the routine could not create the space, then it returns a null pointer.

cnfCreim
Create a temporary C string and import a FORTRAN string into it discarding trailing blanks

Description:

Create a temporary C string, import a FORTRAN string into it and return a pointer to this C string. Any trailing blanks in the FORTRAN string are discarded. The length of the C string that is created is just long enough to hold the FORTRAN string (less trailing blanks), plus the null terminator.

Invocation:

```
pointer = cnfCreim( source_f, source_len )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

Returned Value:

char ∗cnfCreim
A pointer to the storage space allocated by this function.

Notes:

If the routine could not create the space, then it returns a null pointer.

cnfCrela
Create a temporary FORTRAN LOGICAL array and return a pointer to it.

Description:

Memory is obtained for a FORTRAN LOGICAL array, of the specified dimensions and a pointer is returned which may be passed from C to a FORTRAN subroutine. The array is not initialised.

Invocation:

```
string_f = cnfCrela( ndims, dims )
```

Arguments:

int ndims (Given)
The number of dimensions of the FORTRAN array

const int $*$dims (Given)
A 1-D array giving the dimensions of the FORTRAN array.

Returned Value:

F77_LOGICAL_TYPE $*$cnfCrefa
A pointer to the storage that has been allocated by this routine.

Notes:

- If the routine could not create the space, then it returns a null pointer.

cnfExpch
Export a C array of char to a FORTRAN string.

Description:

Export a C array of char to a FORTRAN string, copying 'nchars' characters. No characters, are special so this may be used to export an HDS locator which could contain a null character.

Invocation:

```
cnfExpch( source_c, dest_f, nchars )
```

Arguments:

const char ∗source_c (Given)
A pointer to the input C string

char ∗dest_f (Returned via pointer)
A pointer to the output FORTRAN string

int nchars (Given)
The number of characters to be copied from source_c to dest_f

cnfExpla
Export a C int array to a FORTRAN LOGICAL array

Description:

Export a C int array to a FORTRAN LOGICAL array setting appropriate TRUE or FALSE values in the FORTRAN array.

Invocation:

```
cnfExpla( source_c, dest_f, ndims, dims)
```

Arguments:

const int ∗source_c (Given)
A pointer to the input C array

F77_LOGICAL_TYPE ∗dest_f (Returned via pointer)
A pointer to the FORTRAN output array

int ndims (Given)
The number of dimensions in the arrays

const int ∗dims (Given)
A pointer to a 1-D array giving the dimensions of the arrays

cnfExpn
Export a C string to a FORTRAN string, copying a given maximum number of characters

Description:

Export a C string to a FORTRAN string, copying a maximum of ‘max’ characters. If the C string is shorter than the space allocated to the FORTRAN string, then pad it with blanks, even if the whole source string was not copied as it had more than ‘max’ characters. If the C string is longer than the space allocated to the FORTRAN string, then truncate the string.

Invocation:

```
cnfExpn( source_c, max, dest_f, dest_len )
```

Arguments:

const char ∗source_c (Given)
A pointer to the input C string

int max (Given)
The maximum number of character to be copied from source_c to dest_f

char ∗dest_f (Returned via pointer)
A pointer to the FORTRAN output string

int dest_len (Given)
The length of the FORTRAN output string

cnfExprt
Export a C string to a FORTRAN string

Description:

Export a C string to a FORTRAN string. If the C string is shorter than the space allocated to the FORTRAN string, then pad it with blanks. If the C string is longer than the space allocated to the FORTRAN string, then truncate the string.

Invocation:

```
cnfExprt( source_c, dest_f, dest_len )
```

Arguments:

const char ∗source_c (Given)
A pointer to the input C string

char ∗dest_f (Returned via pointer)
A pointer to the output FORTRAN string

int dest_len (Given)
The length of the output FORTRAN string

cnfExprta
Export a C string array to a FORTRAN CHARACTER array

Description:

Export a C string array to a FORTRAN CHARACTER array. A null character is assumed to terminate each C string – it will not be copied. If the C string is shorter than the space allocated to the FORTRAN string, then pad it with blanks. No more than 'dest_len' characters will be copied for each string.

Invocation:

```
cnfExprta( source_c, source_len, dest_f, dest_len, ndims, dims )
```

Arguments:

const char ∗source_c (Given)
A pointer to the input C array

int source_len (Given)
The maximum number of characters in a string of the C array (including terminating null if required). This would be the last declared dimension of a `char` array.

char ∗dest_f (Returned via pointer)
A pointer to the output FORTRAN array

int dest_len (Given)
The declared maximum number of characters in a element of the FORTRAN array

int ndims (Given)
The number of dimensions of the FORTRAN array

const int ∗dims (Given)
A pointer to a 1-D array specifying the dimensions of the FORTRAN array.

Notes:

The C array is treated as an array of strings but it will actually be an array of `char` with one more dimension than the FORTRAN array, the last dimension being source_len. The other dimensions must be as for the FORTRAN array.

cnfExprtap
Export a C array of pointers to char, to a FORTRAN CHARACTER array

Description:

Export a C array of pointers to char to a FORTRAN CHARACTER array. A null character is assumed to terminate each C string – it will not be copied. If the C string is shorter than the space allocated to the FORTRAN string, then pad it with blanks. No more than 'dest_len' characters will be copied for each string.

Invocation:

```
cnfExprtap( source_c, dest_f, dest_len, ndims, dims )
```

Arguments:

char ∗const ∗source_c (Given)
A pointer to the input C array of pointers to char

char ∗dest_f
A pointer to the output FORTRAN array

int dest_len (Given)
The declared maximum number of characters in a element of the FORTRAN array

int ndims (Given)
The number of dimensions of the arrays

const int ∗dims (Given)
A pointer to a 1-D array specifying the dimensions of the arrays.

Notes:

The array of pointers to char is assumed to point to null-terminated strings. The dimensions of the array of pointers and the FORTRAN character array must be the same.

Strictly, the input array should be declared as '`const char ∗const ∗source_c`', but this would not allow non-constant char to be given.

cnfFptr
Convert a C pointer to a FORTRAN pointer

Description:

Given a C pointer, this function returns the equivalent FORTRAN pointer of type F77_POINTER_TYPE. Note that this conversion is only performed if the C pointer has originally been registered (using `cnfRegp`) for use from both C and FORTRAN. All pointers to space allocated by `cnfCalloc` and `cnfMalloc` are automatically registered in this way.

Invocation:

```
fpointer = cnfFptr( cpointer )
```

Arguments:

void ∗cpointer (Given)
The C pointer.

Returned Value:

F77_POINTER_TYPE cnfCptr
The equivalent FORTRAN pointer value.

Notes:

- A value of zero will be returned if the C pointer has not previously been registered for use from both C and FORTRAN, or if a NULL pointer is supplied.

cnfFree
Free allocated space

Description:

Free space allocated by a call to `cnfCalloc`, `cnfCreat`, `cnfCreib`, `cnfCreim` or `cnfMalloc`.

Invocation:

```
cnfFree( pointer )
```

Arguments:

void ∗pointer (Given)

A pointer to the space to be freed.

Notes:

- This function is not simply equivalent to the C free() function, since if the pointer has been registered (using `cnfRegp`) for use by both C and FORTRAN, then it will be unregistered before the space is freed. All pointers to space allocated by `cnfCalloc` and `cnfMalloc` are automatically registered in this way, so `cnfFree` should always be used to free them.
- It is also safe to free unregistered pointers with this function.

cnfFreef
Free a FORTRAN string

Description:

Return the temporary storage space which was allocated by a previous call to `cnfCref` or `cnfCrefa`.

Invocation:

```
cnfFreef( string_f )
```

Arguments:

F77_CHARACTER_ARG_TYPE ∗string_f (Given)

A pointer (as returned by `cnfCref` or `cnfCrefa`) to the string to be freed.

Notes:

- This function will usually be called via the F77 macro F77_FREE_CHARACTER.

cnfImpb
Import a FORTRAN string into a C string, retaining trailing blanks

Description:

Import a FORTRAN string into a C string retaining trailing blanks. The null character is appended to the C string after all of the blanks in the input string.

Invocation:

```
cnfImpb( source_f, source_len, dest_c )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

char ∗dest_c (Returned via pointer)
A pointer to the output C string

Notes:

No check is made that there is sufficient space allocated to the C string to hold the FORTRAN string. It is the responsibility of the programmer to check this.

cnfImpbn
Import no more than max characters from a FORTRAN string into a C string, retaining trailing blanks

Description:

Import a FORTRAN string into a C string, up to a maximum of 'max' characters, retaining trailing blanks. The null character is appended to the C string after all of the blanks in the input string.

Invocation:

```
cnfImpbn( source_f, source_len, max, dest_c )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

int max (Given)
The maximum number of characters to be copied from the input FORTRAN string to the output C string

char ∗dest_c (Returned via pointer)
A pointer to the output C string

Notes:

No check is made that there is sufficient space allocated to the C string to hold the FORTRAN string. It is the responsibility of the programmer to check this.

cnfImpch
Import a FORTRAN string into a C array of char.

Description:

Import a FORTRAN string into a C array of char, copying 'nchars' characters. No characters, are special so this may be used to import an HDS locator which could contain any character.

Invocation:

```
cnfImprt( source_f, nchars, dest_c )
```

Arguments:

const char *source_f (Given)
A pointer to the input FORTRAN string

int nchars (Given)
The number of characters to be copied from source_f to dest_c

char *dest_c (Returned via pointer)
A pointer to the C array of char

Notes:

No check is made that there is sufficient space allocated to the C array to hold the FORTRAN string. It is the responsibility of the programmer to check this.

cnfImpla
Import a FORTRAN LOGICAL array into a C int array

Description:

Import a FORTRAN LOGICAL array into a C int array setting appropriate TRUE or FALSE values in the C array.

Invocation:

```
cnfImpla( source_f, dest_c, ndims, dims)
```

Arguments:

const F77_LOGICAL_TYPE ∗source_f (Given)
A pointer to the input FORTRAN array

int ∗dest_c (Returned via pointer)
A pointer to the output C array

int ndims (Given)
The number of dimensions in the arrays

const int ∗dims (Given)
A pointer to a 1-D array giving the dimensions of the arrays

cnfImpn
Import no more than max characters from a FORTRAN string into a C string

Description:

Import a FORTRAN string into a C string, up to a maximum of 'max' characters discarding trailing blanks. The null character is appended to the C string after the last non-blank character.

Invocation:

```
cnfImpn( source_f, source_len, max, dest_c )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

int max (Given)
The maximum number of characters to be copied from the input FORTRAN string to the output C string

char ∗dest_c (Returned via pointer)
A pointer to the output C string

Notes:

No check is made that there is sufficient space allocated to the C string to hold the FORTRAN string and a terminating null. It is the responsibility of the programmer to check this.

cnfImprt
Import a FORTRAN string into a C string

Description:

Import a FORTRAN string into a C string, discarding trailing blanks. The null character is appended to the C string after the last non-blank character.

Invocation:

```
cnfImprt( source_f, source_len, dest_c )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length of the input FORTRAN string

char ∗dest_c (Returned via pointer)
A pointer to the output C string

Notes:

No check is made that there is sufficient space allocated to the C string to hold the FORTRAN string and a terminating null. It is the responsibility of the programmer to check this.

cnfImprta
Import a FORTRAN CHARACTER array into a C string array.

Description:

Import a FORTRAN CHARACTER array into a C string array, discarding trailing blanks. The null character is appended to the C string after the last non-blank character copied from the FORTRAN string if there is room. No more than 'dest_len' characters will be copied for each string.

Invocation:

```
cnfImprta( source_f, source_len, dest_c, dest_len, ndims, dims )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN array

int source_len (Given)
The declared maximum number of characters in a element of the FORTRAN array

char ∗dest_c (Returned via pointer)
A pointer to the output C array

int dest_len (Given)
The maximum number of characters in an element of the C array (including terminating null if required). This would be the last declared dimension of a char array.

int ndims (Given)
The number of dimensions of the FORTRAN array

const int ∗dims (Given)
A pointer to a 1-D array giving the dimensions of the FORTRAN array.

Notes:

The C array is treated as an array of strings but it will actually be a char array with one more dimension than the FORTRAN array, the last dimension being 'dest_len'. The other dimensions must be as for the FORTRAN array.

cnfImprtap
Import a FORTRAN CHARACTER array into a C array of pointers to char.

Description:

Import a FORTRAN CHARACTER array into a C array of pointers to char, discarding trailing blanks. The pointers must each point to an area of allocated memory at least 'dest_len' characters long. The null character is appended to the C string after the last non-blank character copied from the FORTRAN string if there is room. No more than 'dest_len' characters will be copied for each string.

Invocation:

```
cnfImprtap( source_f, source_len, dest_c, dest_len, ndims, dims )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN array

int source_len (Given)
The declared maximum number of characters in a element of the FORTRAN array

char ∗const ∗dest_c
A pointer to the output C array

int dest_len (Given)
The maximum number of characters to be copied for each string (including terminating null if required).

int ndims (Given)
The number of dimensions of the arrays

const int ∗dims (Given)
A pointer to a 1-D array giving the dimensions of the arrays.

Notes:

The array of pointers and the FORTRAN character array must have the same dimensions.

cnfLenc
Find the length of a C string

Description:

Find the length (*i.e.* position of the last non blank character) in a C string.

Invocation:

```
result = cnfLenc( source_c )
```

Arguments:

const char ∗source_c (Given)
A pointer to the input C string

Returned Value:

int cnfLenc
The length of the input C string

Notes:

This routine follows the FORTRAN convention of counting positions from one, so with an input string of `"ABCD"` the value returned would be 4.

cnfLenf
Find the length of a FORTRAN string

Description:

Find the length (*i.e.* position of the last non blank character) in a FORTRAN string. This is not necessarily the same as the value of source_len as trailing blanks are not counted.

Invocation:

```
result = cnfLenf( source_f, source_len )
```

Arguments:

const char ∗source_f (Given)
A pointer to the input FORTRAN string

int source_len (Given)
The length (including trailing blanks) of the input FORTRAN string

Returned Value:

int cnfLenf
The length (excluding trailing blanks) of the input FORTRAN string.

Notes:

This routine follows the FORTRAN convention of counting positions from one, so with an input string of `'ABCD'` the value returned would be 4.

cnfMalloc
Allocate space that may be accessed from C and FORTRAN

Description:

This function allocates space in the same way as the standard C malloc() function, except that the pointer to the space allocated is automatically registered (using `cnfRegp`) for use from both C and FORTRAN. This means that the returned pointer may subsequently be converted into a FORTRAN pointer of type F77_POINTER_TYPE (using `cnfFptr`), and back into a C pointer (using `cnfCptr`). The contents of the space may therefore be accessed from both languages.

Invocation:

```
cpointer = cnfMalloc( size );
```

Arguments:

size_t size (Given)
The size of the required space.

Returned Value:

void ∗cnfMalloc
A registered pointer to the allocated space, or NULL if the space could not be allocated.

Notes:

- The allocated space should be freed using `cnfFree` when no longer required.

cnfRegp
Register a pointer for use from both C and FORTRAN

Description:

This is a low-level function which will normally only be required if you are implementing your own memory allocation facilities (all memory allocated by `cnfCalloc` and `cnfMalloc` is automatically registered using this function).

The function attempts to register a C pointer so that it may be used from both C and FORTRAN. If successful, registration subsequently allows the pointer to be converted into a FORTRAN pointer of type F77_POINTER_TYPE (using `cnfFptr`), and then back into a C pointer (using `cnfCptr`). These conversions are possible even if the FORTRAN pointer is stored in a shorter data type than the C pointer.

Not all C pointers may be registered, and registration may fail if the FORTRAN version of the pointer is indistinguishable from that of a pointer which has already been registered. In such a case, a new C pointer must be obtained (e.g. by allocating a different region of memory).

Invocation:

```
result = cnfRegp( cpointer )
```

Arguments:

void $*$cpointer (Given)

The C pointer to be registered.

Returned Value:

int cnfRegp

If registration was successful, the function returns 1. If registration was unsuccessful, it returns zero.

Notes:

- If an internal error occurs (e.g. if insufficient memory is available), the function returns -1.

cnfUregp
Unregister a pointer previously registered using `cnfRegp`

Description:

This is a low-level function which will normally only be required if you are implementing your own memory allocation facilities.

The function accepts a C pointer which has previously been registered for use from both C and FORTRAN (using `cnfRegp`) and removes its registration. Subsequently, conversion between the C pointer and its FORTRAN equivalent (and vice versa) will no longer be performed by `cnfFptr` and `cnfCptr`.

Invocation:

```
cnfUregp( cpointer )
```

Arguments:

void $*$cpointer (Given)

The C pointer to be unregistered.

Notes:

- No action occurs (and no error results) if the C pointer has not previously been registered for use from both C and FORTRAN.

H CNF FORTRAN Function Description

CNF_PVAL
Expand a FORTRAN pointer to its full value

Description:

Given a FORTRAN pointer, stored in an INTEGER variable, this function returns the full value of the pointer (on some platforms, this may be longer than an INTEGER). Typically, this is only required when the pointer is used to pass dynamically allocated memory to another routine using the `%VAL` facility.

Invocation:

```
CALL DOIT( ..., %VAL( CNF_PVAL( FPTR ) ), ...  )
```

Arguments:

FPTR = INTEGER (Given)
The FORTRAN pointer value.

Returned Value:

CNF_PVAL
The full pointer value.

Notes:

- The data type of this function will depend on the platform in use and is declared in the include file CNF_PAR.

CNF_CVAL
Convert a Fortran INTEGER into the same type as used in the TRAIL macro.

Description:

When passing dynamically allocated character strings to Fortran or C routines the character string length is passed as a hidden argument after the visible ones (see TRAIL). With some compilers this length is a 64bit long (INTEGER*8), whereas for others it is more typically a 32bit int (INTEGER*4). Using this function avoids the need to know which size is used for the configured compiler.

Invocation:

```
CALL DOIT( %VAL( CNF_PVAL( FPTR ) ),..., %VAL( CNF_CVAL( FINT ) ) )
```

Arguments:

FINT = INTEGER (Given)

The FORTRAN integer value giving the expected length of the strings.

Returned Value:

CNF_CVAL

The string length in the correct type for the configured compiler. Fortran equivalent of the type used by the TRAIL macro.

Notes:

- The data type of this function will depend on the platform in use and is declared in the include file CNF_PAR.
- When mixing calls that pass locally declared character strings and dynamically allocated ones, all the declared strings must preceed all the dynamic ones in the argument list so that the order of the TRAIL arguments is known.

I References

References

[1] American National Standard – Programming Language – FORTRAN (ANSI X3.9-1978, ISO 1539-1980(E)). Publ, American National Standards Institute.

[2] American National Standard for Information Systems – Programming Language – C (ANSI X3.159-1989). Publ, American National Standards Institute.

[3] Banahan, M.F., 1988. The C Book: featuring the draft ANSI C standard. Publ, Addison-Wesley. B