# CFORTRAN : Interfacing C and FORTRAN

## 0 Abstract

This is a draft of the article which will accompany the introduction of CFORTRAN into the CERN PROGRAM LIBRARY, Release 204.

cfortran.h with its accompanying documentation and examples are currently available by anonymous ftp at:

zebra.desy.de [131.169.2.244]

## 1 Introduction

CFORTRAN provides tools to help create an easy-to-use and machine independent interface between C and FORTRAN routines and global data.

FORTRAN routines and global data can now be used as easily from C as they can from FORTRAN. Similarly FORTRAN programmers can now use C routines and global data just as easily as FORTRAN routines and data. In short, even without a deep understanding of the 'foreign' language, programmers can trivially call routines and use data from the 'foreign' language.

The interface which gives programmers the above freedoms is easily created. The source code for the 'foreign' routines and data is not required; a description of the global data and of the routines' arguments and return values is sufficient. The interface can be created with little or no knowledge of any machine/OS/compiler/linker subtleties.

The description of the routines and data is passed as arguments to C preprocessor directives, defined in cfortran.h, which generate the code needed for the interface. Note that there are no special preprocessors or other programs involved in creating the interface.

The complete CFORTRAN package consists of 4 files: the documentation in cfortran.doc, the engine in cfortran.h, examples in cfortest.c and cfortex.for. The single cfortran.h file currently supports the following machines/OS:

VAX VMS or Ultrix, DECstation, Silicon Graphics, IBM RS/6000, Sun, CRAY, Apollo and HP9000.

## 2 The Interface

CFORTRAN's main aim is to provide the 'best' possible interface between the two languages. 'Best' implies that it is complete and easy, perhaps even trivially easy.

Global data is fairly easily interfaced across C and FORTRAN. cfortran.h provides machinery to help make the interface machine independant, but the largest part of the interface, mapping C structures onto FORTRAN COMMON blocks, is left open. CFORTRAN can only, in cfortex.for and cfortest.c, give examples of how this mapping may be done. CFORTRAN also provides routines for C to insert/extract strings to/from FORTRAN COMMON blocks.

CFORTRAN's emphasis is on the interface for routines, where it is quite complete; providing for the most common types of numeric and character arguments and return values. New types are easily added and can even be defined outside of cfortran.h.

C's interface to FORTRAN routines has been designed to mimic the original

FORTRAN call. This places C programmers on equal footing with FORTRAN programmers when calling FORTRAN routines. e.g. C programmers can follow and easily try out the examples of the original documentation written for FORTRAN programmers. The following example, calling HBOOK1, should demonstrate how closely CFORTRAN imitates the original FORTRAN call.

```
           HBOOK1(1,"pT spectrum of pi+",100,0.,5.,0.);     /* C       */
      CALL HBOOK1(1,'pT spectrum of pi+',100,0.,5.,0.)      !  FORTRAN
```

The FORTRAN interface to C routines provides the machinery to convert the limited number of FORTRAN argument types into the large variety given by C. This implies that no C routines are 'out of reach' for FORTRAN, and even exotic arguments such the value of a structure can be passed via a CFORTRAN interface.

## 3 Creating the Interface

The interface described in the brief outline above hopefully meets most requirements. Equally important is the fact that the interface is easily created. This ease is demonstrated by the C interface to CERN's HBOOK library of routines, contained entirely in the C header file hbook.h. The following excerpt shows how the interface to HBOOK1 is defined.

```
/* hbook.h */
#include "cfortran.h"
        :
#define HBOOK1(ID,CHTITLE,NX,XMI,XMA,VMX) \
     CCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT, \
               ID,CHTITLE,NX,XMI,XMA,VMX)
        :
/* end hbook.h */
```

To use the interface to HBOOK, C programmers merely #include "hbook.h" at the top of their source code. Please note that hbook.h was created following only the [admittedly excellent] HBOOK documentation, without even a single peek at any HBOOK source code.

Creating an interface for FORTRAN to C routines is similar. For example, to access C's memcpy routine, the following file would be compiled, and linked with the FORTRAN code and the library containing memcpy.

```
/* string.c */
#include <string.h>                 /* string.h prototypes memcpy() */
#include "cfortran.h"
FCALLSCSUB3(memcpy,MEMCPY,memcpy,PVOID,PVOID,INT)
```

and a FORTRAN call would look like:      CALL MEMCPY(I,J,7)

In reality, we would probably also want to also set up in string.c the other routines described by string.h.

Note that when we compile string.c, FCALLSCSUB3(...), is expanded into a 'wrapper' function, so called because it is wrapped around the original C function and interfaces the format of the original C functions arguments and return value with the format of the FORTRAN call.

## 4 Getting Started

cfortran.doc provides fairly extensive documentation for CFORTRAN, and perhaps more importantly it lists the few simple commands needed to run the examples in cfortest.c and cfortex.for on the variety of machine supported. Although the examples are slightly more complicated than really necessary, they are nonetheless probably the best place to start understanding what CFORTRAN does.

Users of interfaces created with CFORTRAN, e.g. hbook.h, can perhaps get by with the information in this note, but a casual knowledge of the information in cfortran.doc is probably desirable.

Those wishing to create interfaces using CFORTRAN will have to become familiar with most of the information in cfortran.doc, especially the description of the set of types available for routines’ arguments and return values. cfortran.doc also presents many of the constraints that have to be met in order to create an interface that is portable across the variety of machines supported by cfortran.h.

Understanding cfortran.h is required only by those who wish to support a new type of argument or return value or by those who wish to port cfortran.h to another machine.

Burkhard burow@desy.de

```
/* cfortran.doc 3.8 */         /* anonymous ftp@zebra.desy.de */
/* Burkhard Burow  burow@desy.de                 1990 - 1997. */


              cfortran.h :  Interfacing C or C++ and FORTRAN

Supports: Alpha and VAX VMS, Alpha OSF, DECstation and VAX Ultrix, IBM RS/6000,
          Silicon Graphics, Sun, CRAY, Apollo, HP9000, LynxOS, Convex, Absoft,
          f2c, g77, NAG f90, PowerStation Fortran with Visual C++.

C and C++ are generally equivalent as far as cfortran.h is concerned.
Unless explicitly noted otherwise, mention of C implicitly includes C++.
C++ compilers tested include:
  SunOS> CC +p +w      # Clean compiles.
  IRIX>  CC            # Clean compiles.
  IRIX>  CC -fullwarn  # Still some warnings to be overcome.
  GNU>   g++ -Wall     # Compiles are clean, other than warnings for unused
                       #   cfortran.h static routines.

N.B.: The best documentation on interfacing C or C++ and Fortran is in
      the chapter named something like 'Interfacing C and Fortran'
      to be found in the user's guide of almost every Fortran compiler.
      Understanding this information for one or more Fortran compilers
      greatly clarifies the aims and actions of cfortran.h.
      Such a chapter generally also addresses issues orthogonal to cfortran.h,
      for example the order of array indices, the index of the first element,
      as well as compiling and linking issues.


0 Short Summary of the Syntax Required to Create the Interface
--------------------------------------------------------------

e.g. Prototyping a FORTRAN subroutine for C:

/* PROTOCCALLSFSUBn is optional for C, but mandatory for C++. */

                 PROTOCCALLSFSUB2(SUB_NAME,sub_name,STRING,PINT)
#define SUB_NAME(A,B) CCALLSFSUB2(SUB_NAME,sub_name,STRING,PINT, A,B)

                                 ^     -                                       -
       number of arguments _____|     |   STRING   BYTE    PBYTE       BYTEV(..)|
                                  /   |   STRINGV  DOUBLE  PDOUBLE   DOUBLEV(..)|
                                 /    |  PSTRING   FLOAT   PFLOAT     FLOATV(..)|
        types of arguments ____ /     | PNSTRING   INT     PINT         INTV(..)|
                                \     | PPSTRING   LOGICAL PLOGICAL LOGICALV(..)|
                                 \    |  PSTRINGV  LONG    PLONG       LONGV(..)|
                                  \   |   ZTRINGV  SHORT   PSHORT     SHORTV(..)|
                                      |  PZTRINGV  ROUTINE PVOID      SIMPLE    |
                                       -                                       -


e.g. Prototyping a FORTRAN function for C:
/* PROTOCCALLSFFUNn is mandatory for both C and C++. */
PROTOCCALLSFFUN1(INT,FUN_NAME,fun_name,STRING)
#define FUN_NAME(A)  CCALLSFFUN1(FUN_NAME,fun_name,STRING, A)

e.g. calling FUN_NAME from C:    {int a; a = FUN_NAME("hello");}


e.g. Creating a FORTRAN-callable wrapper for
     a C function returning void, with a 7 dimensional integer array argument:
     [Not supported from C++.]
FCALLSCSUB1(csub_name,CSUB_NAME,csub_name,INTVVVVVVV)


e.g. Creating a FORTRAN-callable wrapper for other C functions:
FCALLSCFUN1(STRING,cfun_name,CFUN_NAME,cfun_name,INT)
           [ ^-- BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG, SHORT, VOID
```

```
              are other types returned by functions.       ]


e.g. COMMON BLOCKs:
FORTRAN:                         common /fcb/  v,w,x
                                 character *(13) v, w(4), x(3,2)
C:
typedef struct { char v[13],w[4][13],x[2][3][13]; } FCB_DEF;
#define FCB COMMON_BLOCK(FCB,fcb)
COMMON_BLOCK_DEF(FCB_DEF,FCB);
FCB_DEF FCB;    /* Define, i.e. allocate memory, in exactly one *.c file. */

e.g. accessing FCB in C:          printf("%.13s",FCB.v);
```

# I Introduction
--------------

cfortran.h is an easy-to-use powerful bridge between C and FORTRAN.
It provides a completely transparent, machine independent interface between
C and FORTRAN routines (= subroutines and/or functions) and global data,
i.e. structures and COMMON blocks.

The complete cfortran.h package consists of 4 files: the documentation in
cfortran.doc, the engine cfortran.h, examples in cfortest.c and
cfortex.f/or. [cfortex.for under VMS, cfortex.f on other machines.]

The cfortran.h package continues to be developed. The most recent version is
available via anonymous ftp at zebra.desy.de (131.169.2.244).

The examples may be run using one of the following sets of instructions:

N.B. Unlike earlier versions, cfortran.h 3.0 and later versions
automatically uses the correct ANSI ## or pre-ANSI /**/
preprocessor operator as required by the C compiler.

N.B. As a general rule when trying to determine how to link C and Fortran,
link a trivial Fortran program using the Fortran compilers verbose option,
in order to see how the Fortran compiler drives the linker. e.g.

```
  unix> cat f.f
           END
  unix> f77 -v f.f
  .. lots of info. follows ...
```

N.B. If using a C main(), i.e. Fortran PROGRAM is not entry of the executable,
and if the link bombs with a complaint about
a missing "MAIN" (e.g. MAIN__, MAIN_, f90_main or similar),
then Fortran has hijacked the entry point to the executable
and wishes to call the rest of the executable via "MAIN".
This can usually be satisfied by doing e.g. 'cc -Dmain=MAIN__ ...'
but often kills the command line arguments in argv and argc.
The f77 verbose option, usually -v, may point to a solution.

```
RS/6000> # Users are strongly urged to use f77 -qextname and cc -Dextname
RS/6000> xlf -c -qextname cfortex.f
RS/6000> cc  -c -Dextname cfortest.c
RS/6000> xlf -o cfortest cfortest.o cfortex.o && cfortest

DECFortran> #Only DECstations with DECFortran for Ultrix RISC Systems.
DECFortran> cc -c -DDECFortran cfortest.c
DECFortran> f77 -o cfortest cfortest.o cfortex.f  &&  cfortest

IRIX xxxxxx 5.2 02282015 IP20 mips
MIPS> # DECstations and Silicon Graphics using the MIPS compilers.
MIPS> cc -o cfortest cfortest.c cfortex.f -lI77 -lU77 -lF77  &&  cfortest
MIPS> # Can also let f77 drive linking, e.g.
```

```
MIPS> cc -c cfortest.c
MIPS> f77 -o cfortest cfortest.o cfortex.f  &&  cfortest

Apollo> # Some 'C compiler 68K Rev6.8' break. [See Section II o) Notes: Apollo]
Apollo> f77 -c cfortex.f && cc -o cfortest cfortest.c cfortex.o  &&  cfortest

VMS> define lnk$library sys$library:vaxcrtl
VMS> cc cfortest.c
VMS> fortran cfortex.for
VMS> link/exec=cfortest cfortest,cfortex
VMS> run cfortest

OSF1 xxxxxx V3.0 347 alpha
Alpha/OSF> # Probably better to let cc drive linking, e.g.
Alpha/OSF> f77 -c cfortex.f
Alpha/OSF> cc  -o cfortest cfortest.c cfortex.o -lUfor -lfor -lFutil -lots -lm
Alpha/OSF> cfortest
Alpha/OSF> # Else may need 'cc -Dmain=MAIN__' to let f77 drive linking.

Sun> # Some old cc(1) need a little help. [See Section II o) Notes: Sun]
Sun> f77 -o cfortest cfortest.c cfortex.f -lc -lm  &&  cfortest
Sun> # Some older f77 may require 'cc -Dmain=MAIN_'.

CRAY> cft77 cfortex.f
CRAY> cc -c cfortest.c
CRAY> segldr -o cfortest.e cfortest.o cfortex.o
CRAY> ./cfortest.e

VAX/Ultrix/cc> # For cc on VAX Ultrix only, do the following once to cfortran.h.
VAX/Ultrix/cc> mv cfortran.h cftmp.h && grep -v "^#pragma" <cftmp.h >cfortran.h

VAX/Ultrix/f77> # In the following, 'CC' is either 'cc' or 'gcc -ansi'. NOT'vcc'
VAX/Ultrix/f77> CC -c -Dmain=MAIN_ cfortest.c
VAX/Ultrix/f77> f77 -o cfortest cfortex.f cfortest.o  &&  cfortest

LynxOS> # In the following, 'CC' is either 'cc' or 'gcc -ansi'.
LynxOS> # Unfortunately cc is easily overwhelmed by cfortran.h,
LynxOS> #  and won't compile some of the cfortest.c demos.
LynxOS> f2c -R cfortex.f
LynxOS> CC -Dlynx -o cfortest cfortest.c cfortex.c -lf2c  &&  cfortest

HP9000> # Tested with HP-UX 7.05 B 9000/380 and with A.08.07 A 9000/730
HP9000> # CC may be either 'c89 -Aa' or 'cc -Aa'
HP9000> #    Depending on the compiler version, you may need to include the
HP9000> #    option '-tp,/lib/cpp' or worse, you'll have to stick to the K&R C.
HP9000> #    [See Section II o) Notes: HP9000]
HP9000> # Users are strongly urged to use f77 +ppu and cc -Dextname
HP9000> CC  -Dextname -c cfortest.c
HP9000> f77 +ppu         cfortex.f  -o cfortest cfortest.o && cfortest
HP9000> # Older f77 may need
HP9000> f77 -c cfortex.f
HP9000> CC -o cfortest cfortest.c cfortex.o -lI77 -lF77 && cfortest

HP0000> # If old-style f77 +800 compiled objects are required:
HP9000> # #define hpuxFortran800
HP9000> cc -c -Aa -DhpuxFortran800 cfortest.c
HP9000> f77 +800 -o cfortest cfortex.f

f2c> # In the following, 'CC' is any C compiler.
f2c> f2c -R cfortex.f
f2c> CC -o cfortest -Df2cFortran cfortest.c cfortex.c -lf2c  &&  cfortest

NAGf90> # cfortex.f is distributed with Fortran 77 style comments.
NAGf90> # To convert to f90 style comments do the following once to cfortex.f:
NAGf90> mv cfortex.f cf_temp.f && sed 's/^C/\!/g' cf_temp.f > cfortex.f
NAGf90> # In the following, 'CC' is any C compiler.
NAGf90> CC -c -DNAGf90Fortran cfortest.c
NAGf90> f90 -o cfortest cfortest.o cfortex.f &&  cfortest
```

```
PC> # On a PC with PowerStation Fortran and Visual_C++
PC> cl /c cftest.c
PC> fl32  cftest.obj cftex.for

GNU> # GNU Fortran
GNU> # See Section VI caveat on using 'gcc -traditional'.
GNU> gcc -ansi -Wall -O -c -Df2cFortran cfortest.c
GNU> g77 -ff2c -o cfortest cfortest.o cfortex.f &&  cfortest

AbsoftUNIX> # Absoft Fortran for all UNIX based operating systems.
AbsoftUNIX> # e.g. Linux or Next on Intel or Motorola68000.
AbsoftUNIX> # Absoft f77 -k allows Fortran routines to be safely called from C.
AbsoftUNIX> gcc -ansi -Wall -O -c -DAbsoftUNIXFortran cfortest.c
AbsoftUNIX> f77 -k -o cfortest cfortest.o cfortex.f && cfortest

CLIPPER> # INTERGRAPH CLIX using CLIPPER C and Fortran compilers.
CLIPPER> # N.B. - User, not cfortran.h, is responsible for
CLIPPER> #         f77initio() and f77uninitio() if required.
CLIPPER> #       - LOGICAL values are not mentioned in CLIPPER doc.s,
CLIPPER> #         so they may not yet be correct in cfortran.h.
CLIPPER> #       - K&R mode (-knr or Ac=knr) breaks FLOAT functions
CLIPPER> #         (see CLIPPER doc.s) and cfortran.h does not fix it up.
CLIPPER> #         [cfortran.h ok for old sun C which made the same mistake.]
CLIPPER> acc cfortest.c -c -DCLIPPERFortran
CLIPPER> af77 cfortex.f cfortest.o -o cfortest
```

By changing the SELECTion ifdef of cfortest.c and recompiling one can try out a few dozen different few-line examples.

The benefits of using cfortran.h include:

1. Machine/OS/compiler independent mixing of C and FORTRAN.

2. Identical (within syntax) calls across languages, e.g.

```
C FORTRAN
      CALL HBOOK1(1,'pT spectrum of pi+',100,0.,5.,0.)
/* C*/
           HBOOK1(1,"pT spectrum of pi+",100,0.,5.,0.);
```

3. Each routine need only be set up once in its lifetime. e.g.

```
/* Setting up a FORTRAN routine to be called by C.
   ID,...,VMX are merely the names of arguments.
   These tags must be unique w.r.t. each other but are otherwise arbitrary. */
PROTOCCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT)
#define HBOOK1(ID,CHTITLE,NX,XMI,XMA,VMX)                    \
     CCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT, \
               ID,CHTITLE,NX,XMI,XMA,VMX)
```

4. Source code is NOT required for the C routines exported to FORTRAN, nor for the FORTRAN routines imported to C. In fact, routines are most easily prototyped using the information in the routines' documentation.

5. Routines, and the code calling them, can be coded naturally in the language of choice. C routines may be coded with the natural assumption of being called only by C code. cfortran.h does all the required work for FORTRAN code to call C routines. Similarly it also does all the work required for C to call FORTRAN routines. Therefore:
   - C programmers need not embed FORTRAN argument passing mechanisms into their code.
   - FORTRAN code need not be converted into C code. i.e. The honed and time-honored FORTRAN routines are called by C.

6. cfortran.h is a single ~1700 line C include file; portable to most remaining, if not all, platforms.

7. STRINGS and VECTORS of STRINGS along with the usual simple arguments to routines are supported as are functions returning STRINGS or numbers. Arrays of pointers to strings and values of structures as C arguments, will soon be implemented. After learning the machinery of cfortran.h, users can expand it to create custom types of arguments. [This requires no modification to cfortran.h, all the preprocessor directives required to implement the custom types can be defined outside cfortran.h]

8. cfortran.h requires each routine to be exported to be explicitly set up. While is usually only be done once in a header file it would be best if applications were required to do no work at all in order to cross languages. cfortran.h's simple syntax could be a convenient back-end for a program which would export FORTRAN or C routines directly from the source code.

-----

Example 1 - cfortran.h has been used to make the C header file hbook.h, which then gives any C programmer, e.g. example.c, full and completely transparent access to CERN's HBOOK library of routines. Each HBOOK routine required about 3 lines of simple code in hbook.h. The example also demonstrates how FORTRAN common blocks are defined and used.

```
/* hbook.h */
#include "cfortran.h"
        :
PROTOCCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT)
#define HBOOK1(ID,CHTITLE,NX,XMI,XMA,VMX)                        \
     CCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT, \
               ID,CHTITLE,NX,XMI,XMA,VMX)
        :
/* end hbook.h */

/* example.c */
#include "hbook.h"
        :
typedef struct {
  int lines;
  int status[SIZE];
  float p[SIZE];  /* momentum */
} FAKE_DEF;
#define FAKE COMMON_BLOCK(FAKE,fake)
COMMON_BLOCK_DEF(FAKE_DEF,FAKE);
        :
main ()
{
        :
           HBOOK1(1,"pT spectrum of pi+",100,0.,5.,0.);
/* c.f. the call in FORTRAN:
      CALL HBOOK1(1,'pT spectrum of pi+',100,0.,5.,0.)
*/
        :
  FAKE.p[7]=1.0;
        :
}
```

N.B. i) The routine is language independent.
    ii) hbook.h is machine independent.
   iii) Applications using routines via cfortran.h are machine independent.

-----

Example 2 - Many VMS System calls are most easily called from FORTRAN, but cfortran.h now gives that ease in C.

```
#include "cfortran.h"
```

```
PROTOCCALLSFSUB3(LIB$SPAWN,lib$spawn,STRING,STRING,STRING)
#define LIB$SPAWN(command,input_file,output_file)          \
     CCALLSFSUB3(LIB$SPAWN,lib$spawn,STRING,STRING,STRING, \
                 command,input_file,output_file)

main ()
{
LIB$SPAWN("set term/width=132","","");
}
```

Obviously the cfortran.h command above could be put into a header file along with the description of the other system calls, but as this example shows, it's not much hassle to set up cfortran.h for even a single call.

-----

Example 3 - cfortran.h and the source cstring.c create the cstring.obj library which gives FORTRAN access to all the functions in C's system library described by the system's C header file string.h.

```
C     EXAMPLE.FOR
      PROGRAM EXAMPLE
      DIMENSION I(20), J(30)
        :
      CALL MEMCPY(I,J,7)
        :
      END

/* cstring.c */
#include <string.h>             /* string.h prototypes memcpy() */
#include "cfortran.h"

        :
FCALLSCSUB3(memcpy,MEMCPY,memcpy,PVOID,PVOID,INT)
        :
```

The simplicity exhibited in the above example exists for many but not all machines. Note 4. of Section II ii) details the limitations and describes tools which try to maintain the best possible interface when FORTRAN calls C routines.

-----

## II Using cfortran.h

The user is asked to look at the source files cfortest.c and cfortex.f for clarification by example.

o) Notes:

o Specifying the Fortran compiler
cfortran.h generates interfaces for the default Fortran compiler. The default can be overridden by defining,

```
     . in the code, e.g.           #define    NAGf90Fortran
  OR . in the compile directive, e.g.: unix> cc -DNAGf90Fortran
```

one of the following before including cfortran.h:

```
 NAGf90Fortran   f2cFortran  hpuxFortran  apolloFortran  sunFortran
  IBMR2Fortran  CRAYFortran  mipsFortran     DECFortran  vmsFortran
 CONVEXFortran       PowerStationFortran          AbsoftUNIXFortran
```

This also allows crosscompilation.
If wanted, NAGf90Fortran, f2cFortran, DECFortran and AbsoftUNIXFortran must be requested by the user.

o /**/
cfortran.h (ab)uses the comment kludge /**/ when the ANSI C preprocessor

```
catenation operator ## doesn't exist. In at least MIPS C, this kludge is
sensitive to  blanks surrounding arguments to macros.
  Therefore, for applications using non-ANSI C compilers, the argtype_i,
routine_name, routine_type and common_block_name arguments to the
PROTOCCALLSFFUNn, CCALLSFSUB/FUNn, FCALLSCSUB/FUNn and COMMON_BLOCK macros
--- MUST NOT --- be followed by any white space characters such as
blanks, tabs or newlines.

o LOGICAL
  FORTRAN LOGICAL values of .TRUE. and .FALSE. do not agree with the C
representation of TRUE and FALSE on all machines. cfortran.h does the
conversion for LOGICAL and PLOGICAL arguments and for functions returning
LOGICAL. Users must convert arrays of LOGICALs from C to FORTRAN with the
C2FLOGICALV(array_name, elements_in_array); macro. Similarly, arrays of LOGICAL
values may be converted from the FORTRAN into C representation by using
F2CLOGICALV(array_name, elements_in_array);

  When C passes or returns LOGICAL values to FORTRAN, by default cfortran.h
only makes the minimal changes required to the value. [e.g. Set/Unset the
single relevant bit or do nothing for FORTRAN compilers which use 0 as FALSE
and treat all other values as TRUE.] Therefore cfortran.h will pass LOGICALs
to FORTRAN which do not have an identical representation to .TRUE. or .FALSE.
This is fine except for abuses of FORTRAN/77 in the style of:
       logical l
       if (l .eq. .TRUE.)     ! (1)
instead of the correct:
       if (l .eqv. .TRUE.)    ! (2)
or:
       if (l)                 ! (3)
For FORTRAN code which treats LOGICALs from C in the method of (1),
LOGICAL_STRICT must be defined before including cfortran.h, either in the
code, "#define LOGICAL_STRICT", or compile with "cc -DLOGICAL_STRICT".
There is no reason to use LOGICAL_STRICT for FORTRAN code which does not do (1).
At least the IBM's xlf and the Apollo's f77 do not even allow code along the
lines of (1).

  DECstations' DECFortran and MIPS FORTRAN compilers use different internal
representations for LOGICAL values. [Both compilers are usually called f77,
although when both are installed on a single machine the MIPS' one is usually
renamed. (e.g. f772.1 for version 2.10.)] cc doesn't know which FORTRAN
compiler is present, so cfortran.h assumes MIPS f77. To use cc with DECFortran
define the preprocessor constant 'DECFortran'.
e.g.        i)  cc -DDECFortran -c the_code.c
        or  ii) #define DECFortran  /* in the C code or add to cfortran.h. */

  MIPS f77 [SGI and DECstations], f2c, and f77 on VAX Ultrix treat
.eqv./.neqv. as .eq./.ne.. Therefore, for these compilers, LOGICAL_STRICT is
defined by default in cfortran.h. [The Sun and HP compilers have not been
tested, so they may also require LOGICAL_STRICT as the default.]

o SHORT and BYTE
  They are irrelevant for the CRAY where FORTRAN has no equivalent to C's short.
Similarly BYTE is irrelevant for f2c and for VAX Ultrix f77 and fort. The
author has tested SHORT and BYTE with a modified cfortest.c/cfortex.f on all
machines supported except for the HP9000 and the Sun.

  BYTE is a signed 8-bit quantity, i.e. values are -128 to 127, on all machines
except for the SGI [at least for MIPS Computer Systems 2.0.] On the SGI it is
an unsigned 8-bit quantity, i.e. values are 0 to 255, although the SGI 'FORTRAN
77 Programmers Guide' claims BYTE is signed. Perhaps MIPS 2.0 is dated, since
the DECstations using MIPS 2.10 f77 have a signed BYTE.

  To minimize the difficulties of signed and unsigned BYTE, cfortran.h creates
the type 'INTEGER_BYTE' to agree with FORTRAN's BYTE. Users may define
SIGNED_BYTE or UNSIGNED_BYTE, before including cfortran.h, to specify FORTRAN's
BYTE. If neither is defined, cfortran.h assumes SIGNED_BYTE.

o CRAY
```

```
  The type DOUBLE in cfortran.h corresponds to FORTRAN's DOUBLE PRECISION.
This implies DOUBLE corresponds to C's double on all machines except for the
CRAY where C's double is the same as float and DOUBLE corresponds instead to
C's long double. Therefore when moving a mixed C and FORTRAN to/from the CRAY,
either the C code will have to change, or the FORTRAN code and cfortran.h
commands will have to change.

To help applications move effortlessly to/from the CRAY, cfortran.h defines the
preprocessor constant DOUBLE_PRECISION, which declares variables which are
equivalent to FORTRAN's DOUBLE PRECISION on all machines, including CRAY.

o f2c
  f2c, by default promotes REAL functions to double. cfortran.h does not (yet)
support this, so the f2c -R option must be used to turn this promotion off.

o NAG f90
  The Fortran 77 subset of Fortran 90 is supported. Extending cfortran.h to
interface C with all of Fortran 90 has not yet been examined.
  The NAG f90 library hijacks the main() of any program and starts the user's
program with a call to: void f90_main(void);
While this in itself is only a minor hassle, a major problem arises because
NAG f90 provides no mechanism to access command line arguments.
  At least version 'NAGWare f90 compiler Version 1.1(334)' appended _CB to
common block names instead of the usual _. To fix, add this to cfortran.h:
#ifdef old_NAG_f90_CB_COMMON
#define COMMON_BLOCK                 CFC_  /* for all other Fortran compilers */
#else
#define COMMON_BLOCK(UN,LN)          _(LN,_CB)
#endif

o RS/6000
  Using "xlf -qextname ...", which appends an underscore, '_', to all FORTRAN
external references, requires "cc -Dextname ..." so that cfortran.h also
generates these underscores.
The use of "xlf -qextname" is STRONGLY ENCOURAGED, since it allows for
transparent naming schemes when mixing C and Fortran.

o HP9000
  Using "f77 +ppu      ...", which appends an underscore, '_', to all FORTRAN
external references, requires "cc -Dextname ..." so that cfortran.h also
generates these underscores.
The use of "f77 +ppu"      is STRONGLY ENCOURAGED, since it allows for
transparent naming schemes when mixing C and Fortran.

  At least one release of the HP /lib/cpp.ansi preprocessor is broken and will
go into an infinite loop when trying to process cfortran.h with the
## catenation operator. The K&R version of cfortran.h must then be used and the
K&R preprocessor must be specified. e.g.
                                         HP9000> cc -Aa -tp,/lib/cpp -c source.c
The same problem with a similar solution exists on the Apollo.
An irrelevant error message '0: extraneous name /usr/include' will appear for
each source file due to another HP bug, and can be safely ignored.
e.g. 'cc -v -c -Aa -tp,/lib/cpp cfortest.c' will show that the driver passes
'-I /usr/include' instead of '-I/usr/include' to /lib/cpp

On some machines the above error causes compilation to stop; one must then use
K&R C, as with old HP compilers which don't support function prototyping.
cfortran.h has to be informed that K&R C is to being used, e.g.
HP9000> cc -D__CF__KnR -c source.c

o AbsoftUNIXFortran
By default, cfortran.h follows the default AbsoftUNIXFortran and prepends _C
to each COMMON BLOCK name. To override the cfortran.h behavior
#define COMMON_BLOCK(UN,LN) before #including cfortran.h.
[Search for COMMON_BLOCK in cfortran.h for examples.]

o Apollo
On at least one release, 'C compiler 68K Rev6.8(168)', the default C
```

```
preprocessor, from cc -A xansi or cc -A ansi, enters an infinite loop when
using cfortran.h. This Apollo bug can be circumvented by using:
     . cc -DANSI_C_preprocessor=0 to force use of /**/, instead of '##'.
 AND . The pre-ANSI preprocessor, i.e. use cc -Yp,/usr/lib
The same problem with a similar solution exists on the HP.

o Sun
Old versions of cc(1), say <~1986, may require help for cfortran.h applications:
 . #pragma may not be understood, hence cfortran.h and cfortest.c may require
   sun> mv cfortran.h cftmp.h && grep -v "^#pragma" <cftmp.h >cfortran.h
   sun> mv cfortest.c cftmp.c && grep -v "^#pragma" <cftmp.c >cfortest.c
 . Old copies of math.h may not include the following from a newer math.h.
   [For an ancient math.h on a 386 or sparc, get similar from a new math.h.]
   #ifdef mc68000     /* 5 lines Copyright (c) 1988 by Sun Microsystems, Inc. */
   #define FLOATFUNCTIONTYPE    int
   #define RETURNFLOAT(x)               return (*(int *)(&(x)))
   #define ASSIGNFLOAT(x,y)     *(int *)(&x) = y
   #endif

o CRAY, Sun, Apollo [pre 6.8 cc], VAX Ultrix and HP9000
  Only FORTRAN routines with less than 15 arguments can be prototyped for C,
since these compilers don't allow more than 31 arguments to a C macro. This can
be overcome, [see Section IV], with access to any C compiler without this
limitation, e.g. gcc, on ANY machine.

o VAX Ultrix
  vcc (1) with f77 is not supported. Although:
VAXUltrix> f77 -c cfortex.f
VAXUltrix> vcc -o cfortest cfortest.c cfortex.o -lI77 -lU77 -lF77  &&  cfortest
will link and run. However, the FORTRAN standard I/O is NOT merged with the
stdin and stdout of C, and instead uses the files fort.6 and fort.5. For vcc,
f77 can't drive the linking, as for gcc and cc, since vcc objects must be
linked using lk (1).  f77 -v doesn't tell much, and without VAX Ultrix manuals,
the author can only wait for the info. required.

  fort (1) is not supported. Without VAX Ultrix manuals the author cannot
convince vcc/gcc/cc and fort to generate names of routines and COMMON blocks
that match at the linker, lk (1). i.e. vcc/gcc/cc prepend a single underscore
to external references, e.g. NAME becomes _NAME, while fort does not modify the
references. So ... either fort has prepend an underscore to external
references, or vcc/gcc/cc have to generate unmodified names. man 1 fort
mentions JBL, is JBL the only way?

o VAX VMS C
  The compiler 'easily' exhausts its table space and generates:
%CC-F-BUGCHECK, Compiler bug check during parser phase    .
              Submit an SPR with a problem description.
              At line number 777 in DISK:[DIR]FILE.C;1.
where the line given, '777', includes a call across C and FORTRAN via
cfortran.h, usually with >7 arguments and/or very long argument expressions.
This SPR can be staved off, with the simple modification to cfortran.h, such
that the relevant CCALLSFSUBn (or CCALLSFFUNn or FCALLSCFUNn) is not
cascaded up to CCALLSFSUB14, and instead has its own copy of the contents of
CCALLSFSUB14. [If these instructions are not obvious after examining cfortran.h
please contact the author.]
[Thanks go to Mark Kyprianou (kyp@stsci.edu) for this solution.]

o Mips compilers
  e.g. DECstations and SGI, require applications with a C main() and calls to
GETARG(3F), i.e. FORTRAN routines returning the command line arguments, to use
two macros as shown:
        :
CF_DECLARE_GETARG;              /* This must be external to all routines.     */
        :
main(int argc, char *argv[])
{
        :
CF_SET_GETARG(argc,argv);       /* This must precede any calls to GETARG(3F). */
```

```
        :
}
```
The macros are null and benign on all other systems. Sun's GETARG(3F) also doesn't work with a generic C main() and perhaps a workaround similar to the Mips' one exists.

o Alpha/OSF
Using the DEC Fortran and the DEC C compilers of DEC OSF/1 [RT] V1.2 (Rev. 10), Fortran, when called from C, has occasional trouble using a routine received as a dummy argument.

e.g. In the following the Fortran routine 'e' will crash when it tries to use the C routine 'c' or the Fortran routine 'f'.
The example works on other systems.

```
C FORTRAN                           /* C */
      integer function f()          #include <stdio.h>
      f = 2                         int f_();
      return                        int e_(int (*u)());
      end
                                    int c(){ return 1;}
      integer function e(u)         int d (int (*u)()) { return u();}
      integer u
      external u                    main()
      e=u()                         {         /* Calls to d  work.  */
      return                        printf("d (c ) returns %d.\n",d (c ));
      end                           printf("d (f_) returns %d.\n",d (f_));
                                              /* Calls to e_ crash. */
                                    printf("e_(c ) returns %d.\n",e_(c ));
                                    printf("e_(f_) returns %d.\n",e_(f_));
                                    }
```

Solutions to the problem are welcomed!
A kludge which allows the above example to work correctly, requires an extra argument to be given when calling the dummy argument function.
i.e. Replacing 'e=u()' by 'e=u(1)' allows the above example to work.

o The FORTRAN routines are called using macro expansions, therefore the usual caveats for expressions in arguments apply. The expressions to the routines may be evaluated more than once, leading to lower performance and in the worst case bizarre bugs.

o For those who wish to use cfortran.h in large applications. [See Section IV.]
This release is intended to make it easy to get applications up and running.
This implies that applications are not as efficient as they could be:
- The current mechanism is inefficient if a single header file is used to describe a large library of FORTRAN functions. Code for a static wrapper fn. is generated in each piece of C source code for each FORTRAN function specified with the CCALLSFFUNn statement, irrespective of whether or not the function is ever called.
- Code for several static utility routines internal to cfortran.h is placed into any source code which #includes cfortran.h. These routines should probably be in a library.

## i) Calling FORTRAN routines from C:

The FORTRAN routines are defined by one of the following two instructions:

for a SUBROUTINE:
```
/* PROTOCCALLSFSUBn is optional for C, but mandatory for C++. */
PROTOCCALLSFSUBn(ROUTINE_NAME,routine_name,argtype_1,...,argtype_n)
#define     Routine_name(argname_1,..,argname_n)               \
CCALLSFSUBn(ROUTINE_NAME,routine_name,argtype_1,...,argtype_n, \
                         argname_1,..,argname_n)
```

```
for a FUNCTION:
PROTOCCALLSFFUNn(routine_type,ROUTINE_NAME,routine_name,argtype_1,...,argtype_n)
#define     Routine_name(argname_1,..,argname_n)               \
CCALLSFFUNn(ROUTINE_NAME,routine_name,argtype_1,...,argtype_n, \
                         argname_1,..,argname_n)

Where:
'n' = 0->14 [SUBROUTINE's ->20] (easily expanded in cfortran.h to > 14 [20]) is
    the number of arguments to the routine.
Routine_name = C       name of the routine (IN UPPER CASE LETTERS).[see 2.below]
ROUTINE_NAME = FORTRAN name of the routine (IN UPPER CASE LETTERS).
routine_name = FORTRAN name of the routine (IN lower case LETTERS).
routine_type = the type of argument returned by FORTRAN functions.
             = BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG, SHORT, STRING, VOID.
               [Instead of VOID one would usually use CCALLSFSUBn.
                VOID forces a wrapper function to be used.]
argtype_i    = the type of argument passed to the FORTRAN routine and must be
               consistent in the definition and prototyping of the routine s.a.
             = BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG, SHORT, STRING.
             For vectors, i.e. 1 dim. arrays use
             = BYTEV, DOUBLEV, FLOATV, INTV, LOGICALV, LONGV, SHORTV,
               STRINGV, ZTRINGV.
             For vectors of vectors, i.e. 2 dim. arrays use
             = BYTEVV, DOUBLEVV, FLOATVV, INTVV, LOGICALVV, LONGVV, SHORTVV.
             For n-dim. arrays, 1<=n<=7 [7 is the maximum in Fortran 77],
             = BYTEV..nV's..V, DOUBLEV..V, FLOATV..V, INTV..V, LOGICALV..V,
               LONGV..V, SHORTV..V.
                N.B. Array dimensions and types are checked by the C compiler.
             For routines changing the values of an argument, the keyword is
                  prepended by a 'P'.
             = PBYTE, PDOUBLE, PFLOAT, PINT, PLOGICAL, PLONG, PSHORT,
               PSTRING, PSTRINGV, PZTRINGV.
             For EXTERNAL procedures passed as arguments use
             = ROUTINE.
             For exceptional arguments which require no massaging to fit the
                  argument passing mechanisms use
             = PVOID.
                The argument is cast and passed as (void *).
                Although PVOID could be used to describe all array arguments on
                most (all?) machines , it shouldn't be because the C compiler
                can no longer check the type and dimension of the array.
argname_i    = any valid unique C tag, but must be consistent in the definition
               as shown.

Notes:

1. cfortran.h may be expanded to handle a more argument type. To suppport new
arguments requiring complicated massaging when passed  between Fortran and C,
the user will have to understand cfortran.h and follow its code and mechanisms.

To define types requiring little or no massaging when passed between Fortran
and C, the pseudo argument type SIMPLE may be used.
For a user defined type called 'newtype', the definitions required are:

/* The following 7 lines are required verbatim.
   'newtype' is the name of the new user defined argument type.
*/
#define newtype_cfV(  T,A,B,F)       SIMPLE_cfV(T,A,B,F)
#define newtype_cfSEP(T,  B)         SIMPLE_cfSEP(T,B)
#define newtype_cfINT(N,A,B,X,Y,Z)   SIMPLE_cfINT(N,A,B,X,Y,Z)
#define newtype_cfSTR(N,T,A,B,C,D,E) SIMPLE_cfSTR(N,T,A,B,C,D,E)
#define newtype_cfCC( T,A,B)         SIMPLE_cfCC(T,A,B)
#define newtype_cfAA( T,A,B)         newtype_cfB(T,A) /* Argument B not used. */
#define newtype_cfU(  T,A)           newtype_cfN(T,A)

/* 'parameter_type(A)' is a declaration for 'A' and describes the type of the
parameter expected by the Fortran function.  This type will be used in the
prototype for the function, if  using ANSI C, and to declare the argument used
```

```
by the intermediate function if calling a Fortran FUNCTION.
Valid 'parameter_type(A)' include: int A
                                   void (*A)()
                                   double A[17]
*/
#define newtype_cfN(  T,A)     parameter_type(A)      /* Argument T not used. */

/* Before any argument of the new type is passed to the Fortran routine, it may
be massaged as given by 'massage(A)'.
*/
#define newtype_cfB(  T,A)     massage(A)             /* Argument T not used. */
```

An example of a simple user defined type is given cfortex.f and cfortest.c.
Two uses of SIMPLE user defined types are [don't show the 7 verbatim #defines]:

```
/* Pass the address of a structure, using a type called PSTRUCT */
#define PSTRUCT_cfN(  T,A)        void *A
#define PSTRUCT_cfB(  T,A)       (void *) &(A)

/* Pass an integer by value, (not standard F77 ), using a type called INTVAL */
#define INTVAL_cfN(   T,A)      int A
#define INTVAL_cfB(   T,A)         (A)
```

[If using VAX VMS, surrounding the #defines with "#pragma (no)standard" allows
the %CC-I-PARAMNOTUSED messages to be avoided.]

Upgrades to cfortran.h try to be, and have been, backwards compatible. This compatibility cannot be offered to user defined types. SIMPLE user defined types are less of a risk since they require so little effort in their creation. If a user defined type is required in more than one C header file of interfaces to libraries of Fortran routines, good programming practice, and ease of code maintenance, suggests keeping any user defined type within a single file which is #included as required. To date, changes to the SIMPLE macros were introduced in versions 2.6, 3.0 and 3.2 of cfortran.h.

2. Routine_name is the name of the macro which the C programmer will use in order to call a FORTRAN routine. In theory Routine_name could be any valid and unique name, but in practice, the name of the FORTRAN routine in UPPER CASE works everywhere and would seem to be an obvious choice.

3. <BYTE|DOUBLE|BYTE|DOUBLE|FLOAT|INT|LOGICAL|LONG|SHORT><V|VV|VVV|...>

cfortran.h encourages the exact specification of the type and dimension of array parameters because it allows the C compiler to detect errors in the arguments when calling the routine.

cfortran.h does not strictly require the exact specification since the argument is merely the address of the array and is passed on to the calling routine. Any array parameter could be declared as PVOID, but this circumvents C's compiletime ability to check the correctness of arguments and is therefore discouraged.

Passing the address of these arguments implies that PBYTEV, PFLOATV, ... , PDOUBLEVV, ... don't exist in cfortran.h, since by default the routine and the calling code share the same array, i.e. the same values at the same memory location.

These comments do NOT apply to arrays of (P)S/ZTRINGV. For these parameters, cfortran.h passes a massaged copy of the array to the routine. When the routine returns, S/ZTRINGV ignores the copy, while PS/ZTRINGV replaces the calling code's original array with copy, which may have been modified by the called routine.

4. (P)STRING(V):
- STRING - If the argument is a fixed length character array, e.g. char ar[8];,

the string is blank, ' ', padded on the right to fill out the array before being passed to the FORTRAN routine. The useful size of the string is the same in both languages, e.g. ar[8] is passed as character*7. If the argument is a pointer, the string cannot be blank padded, so the length is passed as strlen(argument). On return from the FORTRAN routine, pointer arguments are not disturbed, but arrays have the terminating '\0' replaced to its original position. i.e. The padding blanks are never visible to the C code.

- PSTRING - The argument is massaged as with STRING before being passed to the FORTRAN routine. On return, the argument has all trailing blanks removed, regardless of whether the argument was a pointer or an array.

- (P)STRINGV - Passes a 1- or 2-dimensional char array. e.g. char a[7],b[6][8];
STRINGV may thus also pass a string constant, e.g. "hiho".
(P)STRINGV does NOT pass a pointer, e.g. char *, to either a 1- or a 2-dimensional array, since it cannot determine the array dimensions.
A pointer can only be passed using (P)ZTRINGV.
N.B. If a C routine receives a character array argument, e.g. char a[2][3], such an argument is actually a pointer and my thus not be passed by (P)STRINGV. Instead (P)ZTRINGV must be used.

- STRINGV - The elements of the argument are copied into space malloc'd, and each element is padded with blanks. The useful size of each element is the same in both languages. Therefore char bb[6][8]; is equivalent to character*7 bb(6). On return from the routine the malloc'd space is simply released.

- PSTRINGV - Since FORTRAN has no trailing '\0', elements in an array of strings are contiguous. Therefore each element of the C array is padded with blanks and strip out C's trailing '\0'. After returning from the routine, the trailing '\0' is reinserted and kill the trailing blanks in each element.

- SUMMARY: STRING(V) arguments are blank padded during the call to the FORTRAN routine, but remain original in the C code. (P)STRINGV arguments are blank padded for the FORTRAN call, and after returning from FORTRAN trailing blanks are stripped off.

5. (P)ZTRINGV:
- (P)ZTRINGV - is identical to (P)STRINGV,
except that the dimensions of the array of strings is explicitly specified, which thus also allows a pointer to be passed.
(P)ZTRINGV can thus pass a 1- or 2-dimensional char array, e.g. char b[6][8], or it can pass a pointer to such an array, e.g. char *p;.
ZTRINGV may thus also pass a string constant, e.g. "hiho".
If passing a 1-dimensional array, routine_name_ELEMS_j (see below) must be 1.
[Users of (P)ZTRINGV should examine cfortest.c for examples.]:

- (P)ZTRINGV must thus be used instead of (P)STRINGV whenever sizeof() can't be used to determine the dimensions of the array of string or strings. e.g. when calling FORTRAN from C with a char * received by C as an argument.

- There is no (P)ZTRING type, since (P)ZTRINGV can pass a 1-dimensional array or a pointer to such an array, e.g. char a[7], *b;
If passing a 1-dimensional array, routine_name_ELEMS_j (see below) must be 1.

- To specify the numbers of elements,
routine_name_ELEMS_j and routine_name_ELEMLEN_j must be defined as shown below before interfacing the routine with CCALLSFSUBn, PROTOCCALLSFFUNn, etc.

```
#define routine_name_ELEMS_j   ZTRINGV_ARGS(k)
                                 [..ARGS for subroutines, ..ARGF for functions.]
or
#define routine_name_ELEMS_j   ZTRINGV_NUM(l)
Where: routine_name is as above.
       j            [1-n], is the argument being specifying.
       k            [1-n], the value of the k'th argument is the dynamic number
                    of elements for argument j. The k'th argument must be
                    of type BYTE, DOUBLE, FLOAT, INT, LONG or SHORT.
```

```
       l            the number of elements for argument j. This must be an
                    integer constant available at compile time.
                    i.e. it is static.

- Similarly to specify the useful length, [i.e. don't count C's trailing '\0',]
of each element:
#define routine_name_ELEMLEN_j ZTRINGV_ARGS(m)
                                 [..ARGS for subroutines, ..ARGF for functions.]
or
#define routine_name_ELEMLEN_j ZTRINGV_NUM(q)
Where: m            [1-n], as for k but this is the length of each element.
       q            as for l but this is the length of each element.


6. ROUTINE
The argument is an EXTERNAL procedure.

When C passes a routine to Fortran, the language of the function must be
specified as follows:  [The case of some_*_function must be given as shown.]

When C passes a C routine to a Fortran:
    FORTRAN_ROUTINE(arg1, .... ,
                    C_FUNCTION(SOME_C_FUNCTION,some_c_function),
                    ...., argn);

and similarly when C passes a Fortran routine to Fortran:
    FORTRAN_ROUTINE(arg1, .... ,
                    FORTRAN_FUNCTION(SOME_FORT_FUNCTION,some_fort_function),
                    ...., argn);

If fcallsc has been redefined; the same definition of fcallsc used when creating
the wrapper for 'some_c_function' must also be defined when C_FUNCTION is used.
See ii) 4. of this section for when and how to redefine fcallsc.

ROUTINE was introduced with cfortran.h version 2.6. Earlier versions of
cfortran.h used PVOID to pass external procedures as arguments. Using PVOID for
this purpose is no longer recommended since it won't work 'as is' for
apolloFortran, hpuxFortran800, AbsoftUNIXFortran.

7. CRAY only:
In a given piece of source code, where FFUNC is any FORTRAN routine,
FORTRAN_FUNCTION(FFUNC,ffunc)
disallows a previous
#define FFUNC(..) CCALLSFSUBn(FFUNC,ffunc,...) [ or CCALLSFFUNn]
in order to make the UPPER CASE FFUNC callable from C.
#define Ffunc(..) ... is OK though, as are obviously any other names.


ii) Calling C routines from FORTRAN:
    --------------------------------

Each of the following two statements to export a C routine to FORTRAN create
FORTRAN 'wrappers', written in C, which must be compiled and linked along with
the original C routines and with the FORTRAN calling code.

FORTRAN callable 'wrappers' may also be created for C macros. i.e. in this
section, the term 'C function' may be replaced by 'C macro'.

for C functions returning void:
FCALLSCSUBn(             Routine_name,ROUTINE_NAME,routine_name,argtype_1,...,argtyp
e_n)

for all other C functions:
FCALLSCFUNn(routine_type,Routine_name,ROUTINE_NAME,routine_name,argtype_1,...,argtyp
e_n)

Where:
'n' = 0->14 (easily expanded to > 14) stands for the number of arguments to the
```

routine.
Routine_name = the C       name of the routine. [see 9. below]
ROUTINE_NAME = the FORTRAN name of the routine (IN UPPER CASE LETTERS).
routine_name = the FORTRAN name of the routine (IN lower case LETTERS).
routine_type = the type of argument returned by C functions.
= BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG, SHORT, STRING, VOID.
[Instead of VOID, FCALLSCSUBn is recommended.]
argtype_i    = the type of argument passed to the FORTRAN routine and must be
consistent in the definition and prototyping of the routine
= BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG, SHORT, STRING.
For vectors, i.e. 1 dim. arrays use
= BYTEV, DOUBLEV, FLOATV, INTV, LOGICALV, LONGV, SHORTV, STRINGV.
For vectors of vectors, 2 dim. arrays use
= BYTEVV, DOUBLEVV, FLOATVV, INTVV, LOGICALVV, LONGVV, SHORTVV.
For n-dim. arrays use
= BYTEV..nV's..V, DOUBLEV..V, FLOATV..V, INTV..V, LOGICALV..V,
LONGV..V, SHORTV..V.
For routines changing the values of an argument, the keyword is
prepended by a 'P'.
= PBYTE, PDOUBLE, PFLOAT, PINT, PLOGICAL, PLONG, PSHORT,
PSTRING, PNSTRING, PPSTRING, PSTRINGV.
For EXTERNAL procedures passed as arguments use
= ROUTINE.
For exceptional arguments which require no massaging to fit the
argument passing mechanisms use
= PVOID.
The argument is cast and passed as (void *).

Notes:

0. For Fortran calling C++ routines, C++ does NOT easily allow support for:
STRINGV.
BYTEVV, DOUBLEVV, FLOATVV, INTVV, LOGICALVV, LONGVV, SHORTVV.
BYTEV..V, DOUBLEV..V, FLOATV..V, INTV..V, LOGICALV..V, LONGV..V, SHORTV..V.
Though there are ways to get around this restriction,
the restriction is not serious since these types are unlikely to be used as
arguments for a C++ routine.

1. FCALLSCSUB/FUNn expect that the routine to be 'wrapped' has been properly
prototyped, or at least declared.

2. cfortran.h may be expanded to handle a new argument type not already among
the above.

3. <BYTE|DOUBLE|BYTE|DOUBLE|FLOAT|INT|LOGICAL|LONG|SHORT><V|VV|VVV|...>

cfortran.h encourages the exact specification of the type and dimension of
array parameters because it allows the C compiler to detect errors in the
arguments when declaring the routine using FCALLSCSUB/FUNn, assuming the
routine to be 'wrapped' has been properly prototyped.

cfortran.h does not strictly require the exact specification since the argument
is merely the address of the array and is passed on to the calling routine.
Any array parameter could be declared as PVOID, but this circumvents
C's compiletime ability to check the correctness of arguments and is therefore
discouraged.

Passing the address of these arguments implies that PBYTEV, PFLOATV, ... ,
PDOUBLEVV, ... don't exist in cfortran.h, since by default the routine and the
calling code share the same array, i.e. the same values at the same memory
location.

These comments do NOT apply to arrays of (P)STRINGV. For these parameters,
cfortran.h passes a massaged copy of the array to the routine. When the routine
returns, STRINGV ignores the copy, while PSTRINGV replaces the calling

code’s original array with copy, which may have been modified by the called routine.

4. (P(N))STRING arguments have any trailing blanks removed before being passed to C, the same holds true for each element in (P)STRINGV. Space is malloc’d in all cases big enough to hold the original string (elements) as well as C’s terminating ’\0’. i.e. The useful size of the string (elements) is the same in both languages. P(N)STRING(V) => the string (elements) will be copied from the malloc’d space back into the FORTRAN bytes. If one of the two escape mechanisms mentioned below for PNSTRING has been used, the copying back to FORTRAN is obviously not relevant.

5. (PN)STRING’s, [NOT PSTRING’s nor (P)STRINGV’s,] behavior may be overridden in two cases. In both cases PNSTRING and STRING behave identically.

a) If a (PN)STRING argument’s first 4 bytes are all the NUL character, i.e. ’\0\0\0\0’ the NULL pointer is passed to the C routine.

b) If the characters of a (PN)STRING argument contain at least one HEX-00, i.e. the NUL character, i.e. C strings’ terminating ’\0’, the address of the string is simply passed to the C routine. i.e. The argument is treated in this case as it would be with PPSTRING, to which we refer the reader for more detail.

Mechanism a) overrides b). Therefore, to use this mechanism to pass the NULL string, "", to C, the first character of the string must obviously be the NUL character, but of the first 4 characters in the string, at least one must not be HEX-00.

Example:
```
C FORTRAN                         /* C */
      character*40 str            #include "cfortran.h"
C Set up a NULL as :              void cs(char *s) {if (s) printf("%s.\n",s);}
C    i)  4 NUL characters.        FCALLSCSUB1(cs,CS,cs,STRING)
C    ii) NULL pointer.
      character*4 NULL
      NULL = CHAR(0)//CHAR(0)//CHAR(0)//CHAR(0)

      data str/’just some string’/

C Passing the NULL pointer to cs.
      call cs(NULL)
C Passing a copy of ’str’ to cs.
      call cs(str)
C Passing address of ’str’ to cs. Trailing blanks NOT killed.
      str(40:) = NULL
      call cs(str)
      end
```

Strings passed from Fortran to C via (PN)STRING must not have undefined contents, otherwise undefined behavior will result, since one of the above two escape mechanisms may occur depending on the contents of the string.

This is not be a problem for STRING arguments, which are read-only in the C routine and hence must have a well defined value when being passed in.

PNSTRING arguments require special care. Even if they are write-only in the C routine, PNSTRING’s above two escape mechanisms require that the value of the argument be well defined when being passed in from Fortran to C. Therefore, unless one or both of PNSTRING’s escape mechanisms are required, PSTRING should be used instead of PNSTRING.
Prior to version 2.8, PSTRING did have the above two escape mechanisms, but they were removed from PSTRING to allow strings with undefined contents to be passed in. PNSTRING behaves like the old PSTRING.
[Thanks go to Paul Dubois (dubios@icf.llnl.gov) for pointing out that PSTRING must allow for strings with undefined contents to be passed in.]

```
Example:
C FORTRAN                          /* C */
      character*10 s,sn            #include "cfortran.h"
                                   void ps(char *s) {strcpy(s,"hello");}
C Can   call ps  with undef. s.    FCALLSCSUB1(ps,PS,ps,PSTRING)
      call ps(s)                   FCALLSCSUB1(ps,PNS,pns,PNSTRING)
      print *,s,'=s'

C Can't call pns with undef. s.
C e.g. If first 4 bytes of s were
C       "\0\0\0\0", ps would try
C       to copy to NULL because
C       of PNSTRING mechanism.
      sn = ""
      call pns(sn)
      print *,sn,'=sn'

      end
```

6. PPSTRING
The address of the string argument is simply passed to the C routine. Therefore the C routine and the FORTRAN calling code share the same string at the same memory location. If the C routine modifies the string, the string will also be modified for the FORTRAN calling code.
The user is responsible for negociating the differences in representation of a string in Fortran and in C, i.e. the differences are not automatically resolved as they are for (P(N)STRING(V).
This mechanism is provided for two reasons:
- Some C routines require the string to exist at the given memory location, after the C routine has exited. Recall that for the usual (P(N)STRING(V) mechanism, a copy of the FORTRAN string is given to the C routine, and this copy ceases to exist after returning to the FORTRAN calling code.
- This mechanism can save runtime CPU cycles over (P(N)STRING(V), since it does not perform their malloc, copy and kill trailing blanks of the string to be passed.
  Only in a small minority of cases does the potential benefit of the saved CPU cycles outweigh the programming effort required to manually resolve the differences in representation of a string in Fortran and in C.

For arguments passed via PPSTRING, the argument passed may also be an array of strings.

7. ROUTINE
ANSI C requires that the type of the value returned by the routine be known, For all ROUTINE arguments passed from Fortran to C, the type of ROUTINE is specified by defining a cast as follows:

```
#undef  ROUTINE_j
#define ROUTINE_j   (cast)
where:
       j            [1-n], is the argument being specifying.
       (cast)       is a cast matching that of the argument expected by the C
                    function protoytpe for which a wrapper is being defined.
```

e.g. To create a Fortran wrapper for qsort(3C):

```
#undef  ROUTINE_4
#define ROUTINE_4 (int (*)(void *,void *))
FCALLSCSUB4(qsort,FQSORT,fqsort,PVOID,INT,INT,ROUTINE)
```

In order to maintain backward compatibility, cfortran.h defines a generic cast for ROUTINE_1, ROUTINE_2, ..., ROUTINE_14. The user's definition is therefore strictly required only for DEC C, which at the moment is the only compiler which insists on the correct cast for pointers to functions.

When using the ROUTINE argument inside some Fortran code:
- it is difficult to pass a C routine as the parameter,

```
  since in many Fortran implementations,
  Fortran has no access to the normal C namespace.
  e.g. For most UNIX,
       Fortran implicitly only has access to C routines ending in _.
  If the calling Fortran code receives the routine as a parameter
  it can of course easily pass it along.
- if a Fortran routine is passed directly as the parameter,
  the called C routine must call the parameter routine
  using the Fortran argument passing conventions.
- if a Fortran routine is to be passed as the parameter,
  but if Fortran can be made to pass a C routine as the parameter,
  then it may be best to pass a C-callable wrapper for the Fortran routine.
  The called C routine is thus spared all Fortran argument passing conventions.
  cfortran.h can be used to create such a C-callable wrapper
  to the parameter Fortran routine.

ONLY PowerStationFortran:
This Fortran provides no easy way to pass a Fortran routine as an argument to a
C routine. The problem arises because in Fortran the stack is cleared by the
called routine, while in C/C++ it is cleared by the caller.
The C/C++ stack clearing behavior can be changed to that of Fortran by using
stdcall__ in the function prototype. The stdcall__ cannot be applied in this
case since the called C routine expects the ROUTINE parameter to be a C routine
and does not know that it should apply stdcall__.
In principle the cfortran.h generated Fortran callable wrapper for the called C
routine should be able to massage the ROUTINE argument such that stdcall__ is
performed, but it is not yet known how this could be easily done.


8. THE FOLLOWING INSTRUCTIONS ARE NOT REQUIRED FOR VAX VMS
                                     ------------
(P)STRINGV information [NOT required for VAX VMS]: cfortran.h cannot convert
the FORTRAN vector of STRINGS to the required C vector of STRINGS without
explicitly knowing the number of elements in the vector. The application must
do one of the following for each (P)STRINGV argument in a routine before that
routine's FCALLSCFUNn/SUBn is called:

#define routine_name_STRV_Ai NUM_ELEMS(j)
 or
#define routine_name_STRV_Ai NUM_ELEM_ARG(k)
 or
#define routine_name_STRV_Ai TERM_CHARS(l,m)

where: routine_name     is as above.
       i [i=1->n.]      specifies the argument number of a STRING VECTOR.
       j                would specify a fixed number of elements.
       k [k=1->n. k!=i] would specify an integer argument which specifies the
                        number of elements.
       l [char]         the terminating character at the beginning of an
                        element, indicating to cfortran.h that the preceding
                        elements in the vector are the valid ones.
       m [m=1-...]      the number of terminating characters required to appear
                        at the beginning of the terminating string element.
                        The terminating element is NOT passed on to
                        the C routine.

e.g.      #define ce_STRV_A1 TERM_CHARS(' ',2)
          FCALLSCSUB1(ce,CE,ce,STRINGV)

cfortran.h will pass on all elements, in the 1st and only argument to the C
routine ce, of the STRING VECTOR until, but not including, the first string
element beginning with 2 blank, ' ', characters.


9. INSTRUCTIONS REQUIRED ONLY FOR FORTRAN COMPILERS WHICH GENERATE
                 -------------
   ROUTINE NAMES WHICH ARE UNDISTINGUISHABLE FROM C ROUTINE NAMES
   i.e. VAX VMS
```

```
        AbsoftUNIXFortran
        HP9000      if not using the +ppu      option of f77
        IBM RS/6000 if not using the -qextname option of xlf
   Call them the same_namespace compilers.
```

FCALLSCSUBn(...) and FCALLSCFUNn(...), when compiled, are expanded into 'wrapper' functions, so called because they wrap around the original C functions and interface the format of the original C functions' arguments and return values with the format of the FORTRAN call.

Ideally one wants to be able to call the C routine from FORTRAN using the same name as the original C name. This is not a problem for FORTRAN compilers which append an underscore, '_', to the names of routines, since the original C routine has the name 'name', and the FORTRAN wrapper is called 'name_'. Similarly, if the FORTRAN compiler generates upper case names for routines, the original C routine 'name' can have a wrapper called 'NAME', [Assuming the C routine name is not in upper case.] For these compilers, e.g. Mips, CRAY, IBM RS/6000 'xlf -qextname', HP-UX 'f77 +ppu', the naming of the wrappers is done automatically.

For same_namespace compilers things are not as simple, but cfortran.h tries to provide tools and guidelines to minimize the costs involved in meeting their constraints. The following two options can provide same_namespace compilers with distinct names for the wrapper and the original C function.

These compilers are flagged by cfortran.h with the CF_SAME_NAMESPACE constant, so that the change in the C name occurs only when required.

For the remainder of the discussion, routine names generated by FORTRAN compilers are referred to in lower case, these names should be read as upper case for the appropriate compilers.

HP9000: (When f77 +ppu is not used.)
f77 has a -U option which forces uppercase external names to be generated. Unfortunately, cc does not handle recursive macros. Hence, if one wished to use -U for separate C and FORTRAN namespaces, one would have to adopt a different convention of naming the macros which allow C to call FORTRAN subroutines. (Functions are not a problem.) The macros are currently the uppercase of the original FORTRAN name, and would have to be changed to lower case or mixed case, or to a different name. (Lower case would of course cause conflicts on many other machines.) Therefore, it is suggested that f77 -U not be used, and instead that Option a) or Option b) outlined below be used.

VAX/VMS:
For the name used by FORTRAN in calling a C routine to be the same as that of the C routine, the source code of the C routine is required. A preprocessor directive can then force the C compiler to generate a different name for the C routine.

```
e.g.                #if defined(vms)
                    #define name name_
                    #endif
                    void name() {printf("name: was called.\n");}
                    FCALLSCSUB0(name,NAME,name)
```

In the above, the C compiler generates the original routine with the name 'name_' and a wrapper called 'NAME'. This assumes that the name of the routine, as seen by the C programmer, is not in upper case. The VAX VMS linker is not case sensitive, allowing cfortran.h to export the upper case name as the wrapper, which then doesn't conflict with the routine name in C. Since the IBM, HP and AbsoftUNIXFortran platforms have case sensitive linkers this technique is not available to them.

The above technique is required even if the C name is in mixed case, see Option a) for the other compilers, but is obviously not required when Option b) is used.

Option a) Mixed Case names for the C routines to be called by FORTRAN.

If the original C routines have mixed case names, there are no name space conflicts.

Nevertheless for VAX/VMS, the technique outlined above must also used.

Option b) Modifying the names of C routines when used by FORTRAN:

The more robust naming mechanism, which guarantees portability to all machines, 'renames' C routines when called by FORTRAN. Indeed, one must change the names on same_namespace compilers when FORTRAN calls C routines for which the source is unavailable. [Even when the source is available, renaming may be preferable to Option a) for large libraries of C routines.]

Obviously, if done for a single type of machine, it must be done for all machines since the names of routines used in FORTRAN code cannot be easily redefined for different machines.

The simplest way to achieve this end is to do explicitly give the modified FORTRAN name in the FCALLSCSUBn(...) and FCALLSCFUNn(...) declarations. e.g.

```
FCALLSCSUB0(name,CFNAME,cfname)
```

This allows FORTRAN to call the C routine 'name' as 'cfname'. Any name can of course be used for a given routine when it is called from FORTRAN, although this is discouraged due to the confusion it is sure to cause.  e.g. Bizarre, but valid and allowing C's 'call_back' routine to be called from FORTRAN as 'abcd':

```
FCALLSCSUB0(call_back,ABCD,abcd)
```

cfortran.h also provides preprocessor directives for a systematic 'renaming' of the C routines when they are called from FORTRAN. This is done by redefining the fcallsc macro before the FCALLSCSUB/FUN/n declarations as follows:

```
#undef  fcallsc
#define fcallsc(UN,LN) preface_fcallsc(CF,cf,UN,LN)

FCALLSCSUB0(hello,HELLO,hello)
```

Will cause C's routine 'hello' to be known in FORTRAN as 'cfhello'. Similarly all subsequent FCALLSCSUB/FUN/n declarations will generate wrappers to allow FORTRAN to call C with the C routine's name prefaced by 'cf'. The following has the same effect, with subsequent FCALLSCSUB/FUN/n's appending the modifier to the original C routines name.

```
#undef  fcallsc
#define fcallsc(UN,LN) append_fcallsc(Y,y,UN,LN)

FCALLSCSUB0(Xroutine,ROUTINE,routine)
```

Hence, C's Xroutine is called from FORTRAN as:

```
       CALL XROUTINEY()
```

The original behavior of FCALLSCSUB/FUN/n, where FORTRAN routine names are left identical to those of C, is returned using:

```
#undef  fcallsc
#define fcallsc(UN,LN) orig_fcallsc(UN,LN)
```

In C, when passing a C routine, i.e. its wrapper, as an argument to a FORTRAN routine, the FORTRAN name declared is used and the correct fcallsc must be in effect. E.g. Passing 'name' and 'routine' of the above examples to the FORTRAN routines, FT1 and FT2, respectively:

```
/* This might not be needed if fcallsc is already orig_fcallsc. */
#undef  fcallsc
#define fcallsc(UN,LN) orig_fcallsc(UN,LN)
FT1(C_FUNCTION(CFNAME,cfname));

#undef  fcallsc
#define fcallsc(UN,LN) append_fcallsc(Y,y,UN,LN)
FT1(C_FUNCTION(XROUTINE,xroutine));
```

If the names of C routines are modified when used by FORTRAN, fcallsc would usually be defined once in a header_file.h for the application. This definition would then be used and be valid for the entire application and fcallsc would at no point need to be redefined.

```
ONCE AGAIN: THE DEFINITIONS, INSTRUCTIONS, DECLARATIONS AND DIFFICULTIES
DESCRIBED HERE, NOTE 9. of II ii),
APPLY ONLY FOR VAX VMS,
               IBM RS/6000 WITHOUT THE -qextname OPTION FOR xlf, OR
               HP-UX       WITHOUT THE +ppu      OPTION FOR f77
               AbsoftUNIXFortran
AND APPLY ONLY WHEN CREATING WRAPPERS WHICH ENABLE FORTRAN TO CALL C ROUTINES.
```

iii) Using C to manipulate FORTRAN COMMON BLOCKS:
-------------------------------------------------------

FORTRAN common blocks are set up with the following three constructs:

1.
```
#define Common_block_name COMMON_BLOCK(COMMON_BLOCK_NAME,common_block_name)
```

Common_block_name is in UPPER CASE.
COMMON_BLOCK_NAME is in UPPER CASE.
common_block_name is in lower case.
[Common_block_name actually follows the same 'rules' as Routine_name in Note 2. of II i).] This construct exists to ensure that C code accessing the common block is machine independent.

2.
```
COMMON_BLOCK_DEF(TYPEDEF_OF_STRUCT, Common_block_name);
```

where
typedef { ... } TYPEDEF_OF_STRUCT;
declares the structure which maps on to the common block. The #define of Common_block_name must come before the use of COMMON_BLOCK_DEF.

3.
In exactly one of the C source files, storage should be set aside for the common block with the definition:

```
TYPEDEF_OF_STRUCT  Common_block_name;
```

The above definition may have to be omitted on some machines for a common block which is initialized by Fortran BLOCK DATA or is declared with a smaller size in the C routines than in the Fortran routines.

The rules for common blocks are not well defined when linking/loading a mixture of C and Fortran, but the following information may help resolve problems.

From the 2nd or ANSI ed. of K&R C, p.31, last paragraph:
i)
An external variable must be defined, exactly once, outside of any function; this sets aside storage for it.
ii)

```
 The variable must also be declared in each function that wants to access it;
 ...
 The declaration ... may be implicit from context.

In Fortran, every routine says 'common /bar/ foo',
i.e. part ii) of the above, but there's no part i) requirement.
cc/ld on some machines don't require i) either.
Therefore, when handling Fortran, and sometimes C,
the loader/linker must automagically set aside storage for common blocks.

Some loaders, including at least one for the CRAY, turn off the
'automagically set aside storage' capability for Fortran common blocks,
if any C object declares that common block.
Therefore, C code should define, i.e. set aside storage,
for the the common block as shown above.

e.g.
C Fortran
      common /fcb/  v,w,x
      character *(13) v, w(4), x(3,2)

/* C */
typedef struct { char v[13],w[4][13],x[2][3][13]; } FCB_DEF;
#define Fcb COMMON_BLOCK(FCB,fcb)
COMMON_BLOCK_DEF(FCB_DEF,Fcb);
FCB_DEF Fcb;      /* Definition, which sets aside storage for Fcb, */
                  /* may appear in at most one C source file.      */


C programs can place a string (or a multidimensional array of strings) into a
FORTRAN common block using the following call:

C2FCBSTR( CSTR, FSTR,DIMENSIONS);

where:

CSTR is a pointer to the first element of C's copy of the string (array).
     The C code must use a duplicate of, not the original, common block string,
     because the FORTRAN common block does not allocate space for C strings'
     terminating '\0'.

FSTR is a pointer to the first element of the string (array) in the common
     block.

DIMENSIONS is the number of dimensions of string array.
     e.g. char a[10]      has DIMENSIONS=0.
          char aa[10][17] has DIMENSIONS=1.
          etc...

C2FCBSTR will copy the string (array) from CSTR to FSTR, padding with blanks,
' ', the trailing characters as required. C2FCBSTR uses DIMENSIONS and FSTR to
determine the lengths of the individual string elements and the total number of
elements in the string array.

Note that:
- the number of string elements in CSTR and FSTR are identical.
- for arrays of strings, the useful lengths of strings in CSTR and FSTR must be
  the same. i.e. CSTR elements each have 1 extra character to accommodate the
  terminating '\0'.
- On most non-ANSI compilers, the DIMENSION argument cannot be prepended by any
  blanks.


FCB2CSTR( FSTR, CSTR,DIMENSIONS)

is the inverse of C2FCBSTR, and shares the same arguments and caveats.
FCB2CSTR copies each string element of FSTR to CSTR, minus FORTRAN strings'
trailing blanks.
```

cfortran.h USERS ARE STRONGLY URGED TO EXAMINE THE COMMON BLOCK EXAMPLES IN cfortest.c AND cfortex.f. The use of strings in common blocks is demonstrated, along with a suggested way for C to imitate FORTRAN EQUIVALENCE'd variables.

===> USERS OF CFORTRAN.H NEED READ NO FURTHER <===

## III Some Musings

cfortran.h is simple enough to be used by the most basic of applications, i.e. making a single C/FORTRAN routine available to the FORTRAN/C programmers. Yet cfortran.h is powerful enough to easily make entire C/FORTRAN libraries available to FORTRAN/C programmers.

cfortran.h is the ideal tool for FORTRAN libraries which are being (re)written in C, but are to (continue to) support FORTRAN users. It allows the routines to be written in 'natural C', without having to consider the FORTRAN argument passing mechanisms of any machine. It also allows C code accessing these rewritten routines, to use the C entry point. Without cfortran.h, one risks the perverse practice of C code calling a C function using FORTRAN argument passing mechanisms!

Perhaps the philosophy and mechanisms of cfortran.h could be used and extended to create other language bridges such as ADAFORTRAN, CPASCAL, COCCAM, etc.

The code generation machinery inside cfortran.h, i.e. the global structure is quite good, being clean and workable as seen by its ability to meet the needs and constraints of many different compilers. Though the individual instructions of the A..., C..., T..., R... and K... tables deserve to be cleaned up.

## IV Getting Serious with cfortran.h

cfortran.h is set up to be as simple as possible for the casual user. While this ease of use will always be present, 'hooks', i.e. preprocessor directives, are required in cfortran.h so that some of the following 'inefficiencies' can be eliminated if they cause difficulties:

o cfortran.h contains a few small routines for string manipulation. These routines are declared static and are included and compiled in all source code which uses cfortran.h. Hooks should be provided in cfortran.h to create an object file of these routines, allowing cfortran.h to merely prototypes these routines in the application source code. This is the only 'problem' which afflicts both halves of cfortran.h. The remaining discussion refers to the C calls FORTRAN half only.

o Similar to the above routines, cfortran.h generates code for a 'wrapper' routine for each FUNCTION exported from FORTRAN. Again cfortran.h needs preprocessor directives to create a single object file of these routines, and to merely prototype them in the applications.

o Libraries often contain hundreds of routines. While the preprocessor makes quick work of generating the required interface code from cfortran.h and the application.h's, it may be convenient for very large stable libraries to have final_application.h's which already contain the interface code, i.e. these final_application.h's would not require cfortran.h. [The convenience can be imagined for the VAX VMS CC compiler which has a fixed amount of memory for preprocessor directives. Not requiring cfortran.h, with its hundreds of directives, could help prevent this compiler from choking on its internal limits quite so often.]

With a similar goal in mind, cfortran.h defines 100's of preprocessor directives. There is always the potential that these will clash with other tags in the users code, so final_applications.h, which don't require cfortran.h, also provide the solution.

In the same vein, routines with more than 14 arguments can not be interfaced by cfortran.h with compilers which limit C macros to 31 arguments. To resolve this difficulty, final_application.h's can be created on a compiler without this limitation.

Therefore, new machinery is required to do:

```
application.h + cfortran.h => final_application.h
```

The following example may help clarify the means and ends:

If the following definition of the HBOOK1 routine, the /*commented_out_part*/, is passed through the preprocessor [perhaps #undefing and #defining preprocessor constants if creating an application.h for compiler other than that of the preprocessor being used, e.g. cpp -Umips -DCRAY ... ] :

```
#include "cfortran.h"
PROTOCCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT)
/*#define HBOOK1(ID,CHTITLE,NX,XMI,XMA,VMX)                 \*/
     CCALLSFSUB6(HBOOK1,hbook1,INT,STRING,INT,FLOAT,FLOAT,FLOAT, \
                 ID,CHTITLE,NX,XMI,XMA,VMX)
```

A function prototype is produced by the PROTOCCALLSFSUB6(...). Interface code is produced, based on the 'variables', ID,CHTITLE,NX,XMI,XMA,VMX, which will correctly massage a HBOOK1 call. Therefore, adding the #define line:

```
'prototype code'
#define HBOOK1(ID,CHTITLE,NX,XMI,XMA,VMX)                  \
 'interface code'(ID,CHTITLE,NX,XMI,XMA,VMX)
```

which is placed into final_application.h.

The only known limitation of the above method does not allow the 'variable' names to include B1,B2,...,B9,BA,BB,...

Obviously the machinery to automatically generate final_applications.h from cfortran.h and applications.h needs more than just some preprocessor directives, but a fairly simple unix shell script should be sufficient. Any takers?

## V Machine Dependencies of cfortran.h

------------------------------------

Porting cfortran.h applications, e.g. the hbook.h and cstring.c mentioned above, to other machines is trivial since they are machine independent. Porting cfortran.h requires a solid knowledge of the new machines C preprocessor, and its FORTRAN argument passing mechanisms. Logically cfortran.h exists as two halves, a "C CALLS FORTRAN" and a "FORTRAN CALLS C" utility. In some cases it may be perfectly reasonable to port only 'one half' of cfortran.h onto a new system.

The lucky programmer porting cfortran.h to a new machine, must discover the FORTRAN argument passing mechanisms. A safe starting point is to assume that variables and arrays are simply passed by reference, but nothing is guaranteed. Strings, and n-dimensional arrays of strings are a different story. It is doubtful that any systems do it quite like VAX VMS does it, so that a UNIX or f2c versions may provide an easier starting point.

cfortran.h uses and abuses the preprocessor's ## operator. Although the ## operator does not exist in many compilers, many kludges do. cfortran.h uses /**/ with no space allowed between the slashes, '/', and the macros or tags to be concatenated. e.g.

```
#define concat(a,b) a/**/b   /* works*/
main()
{
  concat(pri,ntf)("hello");           /* e.g. */
}
```

N.B. On some compilers without ##, /**/ may also not work. The author may be able to offer alternate kludges.

## VI Bugs in vendors C compilers and other curiosities

1. ULTRIX xxxxxx 4.3 1 RISC

Condolences to long suffering ultrix users!
DEC supplies a working C front end for alpha/OSF, but not for ultrix.

From K&R ANSI C p. 231:

```
   ultrix> cat cat.c
   #define cat(x, y) x ## y
   #define xcat(x,y) cat(x,y)
   cat(cat(1,2),3)
   xcat(xcat(1,2),3)
   ultrix> cc -E cat.c
   123                  <---- Should be: cat(1,2)3
   123                  <---- Correct.
   ultrix>
```

The problem for cfortran.h, preventing use of -std and -std1:

```
   ultrix> cat c.c
   #define cat(x, y) x ## y
   #define xcat(x,y) cat(x,y)
   #define AB(X) X+X
   #define C(E,F,G)  cat(E,F)(G)
   #define X(E,F,G) xcat(E,F)(G)
   C(A,B,2)
   X(A,B,2)
   ultrix> cc -std1 -E c.c
   2+2
   AB  (2)              <---- ?????????????
   ultrix>
   ultrix> cc -std0 -E c.c
   2+2
   AB(2)                <---- ?????????????
   ultrix>
```

Due to further ultrix preprocessor problems,
for all definitions of definitions with arguments,
cfortran.h >= 3.0 includes the arguments and recommends the same,
even though it is not required by ANSI C.
e.g. Users are advised to do

```
   #define fcallsc(UN,LN) orig_fcallsc(UN,LN)
```

instead of

```
   #define fcallsc        orig_fcallsc
```

since ultrix fails to properly preprocess the latter example.
CRAY used to (still does?) occasionally trip up on this problem.

2. ConvexOS convex C210 11.0 convex

In a program with a C main, output to LUN=6=* from Fortran goes into
$pwd/fort.6 instead of stdout. Presumably, a magic incantation can be called
from the C main in order to properly initialize the Fortran I/O.

3. SunOS 5.3 Generic_101318-69 sun4m sparc

The default data and code alignments produced by cc, gcc and f77 are compatible.
If deviating from the defaults, consistent alignment options must be used
across all objects compiled by cc and f77. [Does gcc provide such options?]

4. SunOS 5.3 Generic_101318-69 sun4m sparc with cc: SC3.0.1 13 Jul 1994
   or equivalently
   ULTRIX 4.4 0 RISC using cc -oldc
   are K&R C preprocessors that suffer from infinite loop macros, e.g.

```
  zedy03> cat src.c
  #include "cfortran.h"
                            PROTOCCALLSFFUN1(INT,FREV,frev, INTV)
  #define FREV(A1)               CCALLSFFUN1(    FREV,frev, INTV, A1)
  /* To avoid the problem, deletete these ---^^^^--- spaces.    */
  main() { static int a[] = {1,2}; FREV(a); return EXIT_SUCCESS; }

  zedy03> cc -c -Xs -v -DMAX_PREPRO_ARGS=31 -D__CF__KnR src.c
  "src.c", line 4: FREV: actuals too long
  "src.c", line 4: FREV: actuals too long
  .... 3427 more lines of the same message
  "src.c", line 4: FREV: actuals too long
  cc : Fatal error in /usr/ccs/lib/cpp
  Segmentation fault (core dumped)
```

5. Older sun C compilers

To link to f77 objects, older sun C compilers require the math.h macros:

```
#define RETURNFLOAT(x)   { union {double _d; float _f; } _kluge; \
                           _kluge._f = (x); return _kluge._d;   }
#define ASSIGNFLOAT(x,y) { union {double _d; float _f; } _kluge; \
                           _kluge._d = (y); x = _kluge._f;      }
```

Unfortunately, in at least some copies of the sun math.h, the semi-colon
for 'float _f;' is left out, leading to compiler warnings.

The solution is to correct math.h, or to change cfortran.h to #define
RETURNFLOAT(x) and ASSIGNFLOAT(x,y) instead of including math.h.

6. gcc version 2.6.3 and probably all other versions as well

Unlike all other C compilers supported by cfortran.h,
'gcc -traditional' promotes to double all functions returning float
as demonstrated bu the following example.

```
/* m.c */
#include <stdio.h>
int main() { FLOAT_FUNCTION d(); float f; f = d(); printf("%f\n",f); return 0; }

/* d.c */
float d() { return -123.124; }

burow[29] gcc -c -traditional d.c
burow[30] gcc -DFLOAT_FUNCTION=float m.c d.o && a.out
0.000000
burow[31] gcc -DFLOAT_FUNCTION=double m.c d.o && a.out
-123.124001
burow[32]
```

```
Thus, 'gcc -traditional' is not supported by cfortran.h.
Support would require the same RETURNFLOAT, etc. macro machinery
present in old sun math.h, before sun gave up the same promotion.
```

VII History and Acknowledgements
--------------------------------

```
1.0 - Supports VAX VMS using C 3.1 and FORTRAN 5.4.                    Oct. '90.
1.0 - Supports Silicon Graphics w. Mips Computer 2.0 f77 and cc.       Feb. '91.
          [Port of C calls FORTRAN half only.]
1.1 - Supports Mips Computer System 2.0 f77 and cc.                    Mar. '91.
          [Runs on at least: Silicon Graphics IRIX 3.3.1
                             DECstations with Ultrix V4.1]
1.2 - Internals made simpler, smaller, faster, stronger.               May  '91.
    - Mips version works on IBM RS/6000, this is now called the unix version.
1.3 - UNIX and VAX VMS versions are merged into a single cfortran.h.   July '91.
    - C can help manipulate (arrays of) strings in FORTRAN common blocks.
    - Dimensions of string arrays arguments can be explicit.
    - Supports Apollo DomainOS 10.2 (sys5.3) with f77 10.7 and cc 6.7.

2.0 - Improved code generation machinery creates K&R or ANSI C.        Aug. '91.
    - Supports Sun, CRAY. f2c with vcc on VAX Ultrix.
    - cfortran.h macros now require routine and COMMON block names in both
      upper and lower case. No changes required to applications though.
    - PROTOCCALLSFSUBn is eliminated, with no loss to cfortran.h performance.
    - Improved tools and guidelines for naming C routines called by FORTRAN.
2.1 - LOGICAL correctly supported across all machines.                 Oct. '91.
    - Improved support for DOUBLE PRECISION on the CRAY.
    - HP9000 fully supported.
    - VAX Ultrix cc or gcc with f77 now supported.
2.2 - SHORT, i.e. INTEGER*2, and BYTE now supported.                   Dec. '91.
    - LOGICAL_STRICT introduced. More compact and robust internal tables.
    - typeV and typeVV for type = BYTE, DOUBLE, FLOAT, INT, LOGICAL, LONG,SHORT.
    - FORTRAN passing strings and NULL pointer to C routines improved.
2.3 - Extraneous arguments removed from many internal tables.          May  '92.
    - Introduce pseudo argument type SIMPLE for user defined types.
    - LynxOS using f2c supported. (Tested with LynxOS 2.0 386/AT.)
2.4 - Separation of internal C and Fortran compilation directives.     Oct. '92.
    - f2c and NAG f90 supported on all machines.
2.5 - Minor mod.s to source and/or doc for HP9000, f2c, and NAG f90.   Nov. '92.
2.6 - Support external procedures as arguments with type ROUTINE.      Dec. '92.
2.7 - Support Alpha VMS. Support HP9000 f77 +ppu                       Jan. '93.
    - Support arrays with up to 7 dimensions.
    - Minor mod. of Fortran NULL to C via (P)STRING.
    - Specify the type of ROUTINE passed from Fortran to C [ANSI C requirement.]
    - Macros never receive a null parameter [RS/6000 requirement.]
2.8 - PSTRING for Fortran calls C no longer provides escape to pass    April'93.
      NULL pointer nor to pass address of original string.
      PNSTRING introduced with old PSTRING's behavior.
      PPSTRING introduced to always pass original address of string.
    - Support Alpha/OSF.
    - Document that common blocks used in C should be declared AND defined.

3.0 - Automagic handling of ANSI ## versus K&R /**/ preprocessor op.   March'95.
    - Less chance of name space collisions between cfortran.h and other codes.
    - SIMPLE macros, supporting user defined types, have changed names.
3.1 - Internal macro name _INT not used. Conflicted with IRIX 5.3.     May  '95.
    - SunOS, all versions, should work out of the box.
    - ZTRINGV_ARGS|F(k) may no longer point to a PDOUBLE or PFLOAT argument.
    - ConvexOS 11.0 supported.
3.2 - __hpux no longer needs to be restricted to MAX_PREPRO_ARGS=31.   Oct. '95.
    - PSTRING bug fixed.
    - ZTRINGV_ARGS|F(k) may not point to a PBYTE,PINT,PLONG or PSHORT argument.
    - (P)ZTRINGV machinery improved. Should lead to fewer compiler warnings.
      (P)ZTRINGV no longer limits recursion or the nesting of routines.
```

```
    - SIMPLE macros, supporting user defined types, have changed slightly.
3.3 - Supports PowerStation Fortran with Visual C++.                   Nov. ’95.
    - g77 should work using f2cFortran, though no changes made for it.
    - (PROTO)CCALLSFFUN10 extended to (PROTO)CCALLSFFUN14.
    - FCALLSCFUN10 and SUB10 extended to FCALLSCFUN14 and SUB14.
3.4 - C++ supported,                                                   Dec. ’95.
      but it required the reintroduction of PROTOCCALLSFSUBn for users.
    - HP-UX f77 +800 supported.
3.5 - Absoft UNIX Fortran supported.                                   Sept.’96.
3.6 - Minor corrections to cfortran.doc.                               Oct. ’96.
    - Fixed bug for 15th argument. [Thanks to Tom Epperly at Aspen Tech.]
    - For AbsoftUNIXFortran, obey default of prepending _C to COMMON BLOCK name.
    - Fortran calling C with ROUTINE argument fixed and cleaned up.
3.7 - Circumvent IBM and HP "null argument" preprocessor warning.      Oct. ’96
3.8 - (P)STRINGV and (P)ZTRINGV can pass a 1- or 2-dim. char array.    Feb. ’97
      (P)ZTRINGV thus effectively also provides (P)ZTRING.
      (P)ZTRINGV accepts a (char *) pointer.
```

[’Support’ implies these and more recent releases of the respective
OS/compilers/linkers can be used with cfortran.h.
Earlier releases may also work.]

Acknowledgements:
- CERN very generously sponsored a week in 1994 for me to work on cfortran.h.
- M.L.Luvisetto (Istituto Nazionale Fisica Nucleare - Centro Nazionale
  Analisi Fotogrammi, Bologna, Italy) provided all the support for the port to
  the CRAY. Marisa’s encouragement and enthusiasm was also much appreciated.
- J.Bunn (CERN) supported the port to PowerStation Fortran with Visual C++.
- Paul Schenk (UC Riverside, CERN PPE/OPAL) in June 1993 extended cfortran.h 2.7
  to have C++ call Fortran. This was the starting point for full C++ in 3.4.
- Tony Goelz of Absoft Corporation ported cfortran.h to Absoft UNIX Fortran.
- Though cfortran.h has been created in my ’copious’ free time, I thank
  NSERC for their generous support of my grad. student and postdoc years.
- Univ.Toronto, DESY, CERN and others have provided time on their computers.



Burkhard Burow
burow@desy.de

P.S. Your comments and questions are welcomed and usually promptly answered.

VAX VMS and Ultrix, Alpha, OSF, Silicon Graphics (SGI), DECstation, Mips RISC,
Sun, CRAY, Convex, IBM RS/6000, Apollo DomainOS, HP, LynxOS, f2c, NAG, Absoft,

PowerStation and Visual C++ are registered trademarks of their respective owners.

/* end: cfortran.doc */