6. Tutorial Section

6.1 Interfacing Fortran and C

Alfred Nathaniel CN/ASD

In this article[1] I shall summarize the experience about calling C from Fortran and vice versa.

Over the past two years the originally Fortran-based library package KUIP has been gradually rewritten in C. (KUIP stands for "Kit for a User Interface Package" and provides the user interface for PAW and other applications from the CERN program library. KUIP has been ported to VMS, VM/CMS, MVS, MS-DOS, Windows/NT, and a large number of Unix flavours.)

The rewrite was not a pure "exercise of style" but it appeared to be the best choice in order to improve the quality of the package. Key elements for a user interface package such as dynamic string handling, recursivity, and function pointers are inherent to the C language. Using C allowed the removal of most of the limitations with respect to maximum string lengths built into the original Fortran code. It also facilitated the interface to Motif which is only accessible through C libraries.

The Fortran routines of KUIP have been replaced by C functions. The C code emulates the former Fortran entry points and is therefore completely compatible.

When doing inter-language calls between Fortran and C we have to realize that the Fortran part is fixed, and that the C code has to adapt to the characteristics employed by the Fortran compiler. The `cfortran.h`[2] header file by Burkhard Burow provides convenient preprocessor macros for interfacing Fortran and C. For reasons explained below it does, however, not cover the IBM mainframe systems. Also, `cfortran.h` exercises `cpp` to the full which always involves the risk of stretching the C preprocessor beyond its limits.

By no means we do want to discourage the use of `cfortran.h`. However, if `cfortran.h` cannot be used (because the target platform is not supported or the overhead is considered too large), here you should find the necessary information for doing the Fortran-C interface "by hand".

1- Type Definitions

It is good style to use `typedef` names to map the Fortran data types to their C counterparts. Up to now all platforms are satisfied by a single set of definitions:

```
typedef int     INTEGER;
typedef INTEGER LOGICAL;
typedef float   REAL;
typedef double  DBLPREC;
typedef struct { REAL re; REAL im; } COMPLEX;
```

but, for example, a Fortran compiler with `AUTODBL` option for generation 64-bit code would require different mappings.

2- External Identifiers Naming

First of all, there are differences in the naming of external identifiers (SUBROUTINE, FUNCTION and COMMON block names in Fortran and their C equivalent).

2.1- SUBROUTINE

For a Fortran `CALL SUB` the corresponding C routine has to be named:

`SUB`	all upper case on Cray with `cft77` compiler
`sub`	all lower case on Apollo with `ftn` compiler case insensitive on IBM/370 and VMS
`sub_`	lower case with underscore added on all other system

The HP/UX and RS-6000 Fortran compilers need a special switch (`+ppu` and `-qextname`, respectively) in order to generate `sub_` instead of `sub`. CERN program library policy is to use the form with the appended underscore to avoid the risk of name clashes between Fortran user routines and C library functions.

For convenience I decided to define the capitalized form `Sub` as a preprocessor identifier:

```
#define Sub F77_NAME(sub,SUB)
```

where the preprocessor macro `F77_NAME` maps to the appropriate external name:

```
/* Cray, VM/CMS, MVS, VMS */
#define F77_NAME(name,NAME) NAME

/* Apollo FTN */
#define F77_NAME(name,NAME) name

/* others */
#define F77_NAME(name,NAME) name##_
```

The IBM C/370 compiler requires in addition a

```
#pragma linkage(SUB,FORTRAN)
```

in order to use the correct argument passing mechanism which is different between Fortran and C. Unfortunately the `#pragma` cannot be included into the `F77_NAME` macro and must be specified separately.

[1] This information will also be published as an appendix in the KUIP manual.
[2] `cfortran.h` and `f2h` for generating interface definitions from Fortran source code are available as part of the CERN program library distribution.

2.2- FUNCTION

The same rules as for `SUBROUTINE` apply also for `FUNCTION` names. We found, however, that only `INTEGER` and `LOGICAL` `FUNCTION`s can be emulated in C. `REAL FUNCTION`s and others are not guaranteed to work on all platforms.

2.3- COMMON

Fortran `COMMON` blocks can be accessed from C code. Most compilers apply the same rule for generating the external identifier as for routines but up to now we came across two exceptions: the Convex compiler adds an underscore both in front and at the end. The Absoft Fortran compiler for the NeXT uses the plain lower-case name without underscores.

Again for convenience we define the capitalized form `Com` as a preprocessor identifier:

```
#define Com F77_BLOCK(com,COM)
```

where the preprocessor macro expands to the appropriate identifier:

```
/* Convex */
#define F77_BLOCK(name,NAME) _##name##_

/* NeXT */
#define F77_BLOCK(name,NAME) name

/* others */
#define F77_BLOCK(name,NAME) \
         F77_NAME(name,NAME)
```

Naively one would assume that the C structure mapping the `COMMON` block should be a *declaration* (i.e. "`extern struct ...`") since the storage should be allocated by the Fortran side. Some platforms, however, require the form of a *definition*, i.e. leaving out the `extern` keyword.

In order to cope with this difference we define

```
/* Apollo, Convex, Cray */
#define EXTERN

/* others */
#define EXTERN extern
```

One last complication is that on Apollo the C definition has to be given a special attribute. Otherwise the linker allocates the C structure separate from the `COMMON` block it should map.

Again we can define a system-dependent preprocessor macro to cope with this difference:

```
/* Apollo */
#define F77_COMMON(name) name \
       __attribute((__section(name)))

/* others */
#define F77_COMMON(name) name
```

Having these ingredients it is straightforward to translate any Fortran `COMMON` block into a C `struct`. For example,

```
      INTEGER I
      REAL X
      DOUBLE PRECISION D
      COMMON /COMX/ I,X(3,3),D

      CHARACTER*80 CHTEXT(10)
      COMMON /COMC/ CHTEXT
```

is described by

```
#define Comx F77_BLOCK(comx,COMX)

EXTERN struct {
  INTEGER i;
  REAL x[3][3];
  DBLPREC d;
} F77_COMMON(Comx);

#define Comc F77_BLOCK(comc,COMC)

EXTERN struct {
  char text[10][80];
} F77_COMMON(Comc);
```

and the values are accessible as `Comx.i`, `Comx.x[1][2]` for `X(3,2)`, `Comc.text[4][0]` for `CHTEXT(5)(1:1)` etc.

Note that one should not rely on the possibility to extend the `COMMON` block size by changing the dimensions in the C code. It is platform dependent whether

```
EXTERN struct {
  char text[1000][80];
} F77_COMMON(Comc);
```

without changing the Fortran definition accordingly will actually reserve the requested amount of memory for the `COMMON` block

Also one has to be careful with the placement of `DOUBLE PRECISION` variables. They should always be at an even offset from the start of the `COMMON` block because otherwise in a case like

```
      REAL X,Y
      DOUBLE PRECISION D
      COMMON /COM/ X,D,Y
```

it is not guaranteed that it can be mapped as

```
struct {
  REAL x;
  DBLPREC d;
  REAL y;
} ...
```

While the C compiler is allowed to add padding in order to align `d`, the Fortran compiler is obliged to allocate `X` and `D` at consecutive machine words. Many CPUs signal a bus error upon unaligned access to `DOUBLE PRECISION` variables, and compilers usually add protective code and issue a warning message.

One exception is the HP/UX Fortran compiler which silently adds padding in the same way the C compiler does. That makes the Fortran-C interface easier but at the same time violates the rules of Fortran storage association. In principle the same `COMMON` block could be declared in a different routine as

```
      REAL Z
      COMMON /COM/ Z(4)
```

and for the HP/UX Fortran compiler `Y` and `Z(4)` do not refer to the same memory location anymore.

2.4- /PAWC/

Some of the complications in accessing `COMMON` blocks from C could be avoided by defining a structure pointer which is filled by a call from a Fortran routine passing the first variable in the `COMMON` block.

In fact in KUIP this possibility is used for the `/PAWC/` Zebra store. The C structure definition looks like

```
#define Pawc kc_pawc

EXTERN struct COMMON_PAWC {
  INTEGER NWPAR;
  INTEGER IXPAWC;
  INTEGER IHBOOK;
  INTEGER IXHIGZ;
  INTEGER IXKUIP;
  INTEGER IFENCE[5];
  INTEGER LQDATA[8];
  INTEGER IQDATA[999];
} *Pawc;
```

and the pointer is initialized by a `CALL KIPAWC(NWPAR)` to

```
void Kipawc( struct COMMON_PAWC *pawc )
{
  Pawc = pawc;
}
```

Linking to `/PAWC/` in this way was introduced due to a request from W. Wojcik of IN2P3 in Lyon. They wanted to exploit the option `DC(name)` of the VS-Fortran compiler which allows to allocate `COMMON` blocks at execution time. The generated code references `COMMON` block variables indirectly through a pointer to dynamic memory rather than as an absolute address resolved by the loader. (The libraries produced on CERNVM do not make use of the `DC` option. The C code works either way without change.)

Preprocessor macros allow to write C code accessing the Zebra store in a Fortran-ish style. For example, the equivalent of

```
      DO 10 I = 1, IQ(LX-1)
        Q(LX+1) = 3.14
 10   CONTINUE
```

can be written as

```
#define LQ(n)           Pawc->LQDATA[n-1]
#define IQ(n)           Pawc->IQDATA[n-1]
#define  Q(n) (((REAL*)(Pawc->IQDATA))[n-1])

  for( i = 1; i <= IQ(lx-1); i++ ) {
    Q(lx+1) = 3.14;
  }
```

3- Argument Passing in Calling C from Fortran

3.1- Numeric data

Passing numeric arguments between Fortran and C is straightforward. One only has to remember that Fortran always uses *call-by-reference*, i.e. it passes the address of the memory location where the value is stored.

Consequently, the Fortran routine call

```
      CALL SUB(X,2)
```

corresponds to the C function declaration:

```
void Sub( REAL* xp, INTEGER* ip )
```

The value itself is accessed as in

```
  float x = *xp;
```

To store a value back into the Fortran variable `X` one has to write in C

```
  *xp = 3.14;
```

3.2- Arrays

If `X` is an array the third element `X(3)` (in Fortran) has to be addressed as `xp[2]`, since in C array indices start at 0.

It is important to note that the pointers received as arguments **must not** be used as local variable. For example, one might be tempted to write:

```
void Sub( REAL* xp, INTEGER* ip )
{
  int i;
  for( i = *ip; i > 0; i-- ) {
    float x = *xp++;
    ...
  }
}
```

where `xp` is incremented to point to the next element, which is perfectly legal C. But there is a potential problem if `Sub` is called from Fortran.

Some machines (e.g. IBM/370 and Convex) use argument blocks, i.e. the call passes the address of a memory region containing the argument addresses. Since a Fortran routine `SUB` would have no way the alter the content of the argument block, the Fortran compiler treats it as a compile time constant. As a result, only the first call to `Sub` will access the correct array elements.

3.3- LOGICAL

The representation of `.TRUE.` varies between platforms. It seems to be safe to test a `LOGICAL` input against `0` but output parameters should use the proper `.TRUE.` and `.FALSE.` constants. Returning the C values `~0` or `0` will work in tests such as

```
IF(L) ...
```

but may fail for

```
IF(L .EQ. .TRUE.) ...
```

3.4- EXTERNAL

C entry points can receive Fortran `EXTERNAL` arguments as in

```
EXTERNAL XT
CALL SUB(XT)
```

Most platforms pass directly the address of the routine

```
typedef void SUBROUTINE();

void Sub( SUBROUTINE* xt )
```

but some compilers (AIX/370, IBM/370, Apollo, NeXT) store the routine address in a memory cell whose address the called function receives. If we undo the indirection

```
void Sub( SUBROUTINE** xtp )
{
  SUBROUTINE* xt = *xtp;
  ...
}
```

in the routine header we can use afterwards in both cases the same C statement:

```
(*xt)(...)
```

for calling the routine through the function pointer.

Using a `typedef` name does more than just improve the readability of the declaration. For the IBM C/370 compiler it is essential to add

```
#pragma linkage(SUBROUTINE,FORTRAN)
```

in order to declare that function pointers of type `SUBROUTINE` have to use the Fortran convention of argument passing.

3.5- CHARACTER

This is one of the most tricky and tedious part of Fortran/C intermixing. A Fortran `CHARACTER` variable is characterized by two items of information—the address and the length—and the mechanism for passing these two values is compiler dependent. We can label the different mechanisms according to the number of arguments received by the called routine. For a `SUBROUTINE` with `N` parameters in total of which `k` are of type `CHARACTER` the following number of arguments can be passed:

- `N` Address and length are passed in a single string descriptor argument.
- `N+k` The address is passed at the position of the `CHARACTER` argument and for each `CHARACTER` argument the length information is added at the end of the regular argument list.
- `2N` Like the previous case but there is an extra length indicator, for each `CALL` argument.

With variations on the type of the length indicator these are the three different schemes I have come across up to now. Of course, there are more possible permutations a new Fortran compiler may choose from.

To give a more complete picture we will take the example of the following basic Fortran routine

```
SUBROUTINE SUB(X,CH,Y)
REAL X,Y
CHARACTER*(*) CH
```

and give the different headers, according to machines, for the C equivalent function in order to make it Fortran callable without any problem. In each case we will end up in the C function with the fixed variable names `char* ch_ptr` and `int ch_len` containing the address and length of `CH`.

<u>VMS</u>

VMS passes the address of a string descriptor containing the address and length information as structure elements:

```
#include <descrip.h>

void Sub( REAL* x,
          struct dsc$descriptor_s* ch_dsc,
          REAL* y )
{
  char* ch_ptr = ch_dsc->dsc$a_pointer;
  int   ch_len = ch_dsc->dsc$w_length;
  ...
}
```

<u>Cray</u>

Cray packs both items into the 64-bit word received for each argument (and the bit fields used are actually processor dependent):

```
#include <fortran.h>

void Sub( REAL* x, _fcd ch_dsc, REAL* y )
{
  char* ch_ptr = _fcdtocp(ch_dsc);
  int   ch_len = _fcdlen(ch_dsc);
  ...
}
```

VS-Fortran

IBM VS-Fortran uses the `2N` scheme with *call-by-reference* for the length information:

```
#pragma linkage(SUB,FORTRAN)

void Sub( REAL* x,     char* ch_ptr, REAL* y,
     int*  x_size, int* ch_ref, int*  y_size )
{
  int ch_len = *ch_ref;
  ...
}
```

NeXT

For the Absoft Fortran compiler on NeXT it looks similar except that *call-by-value* is used:

```
void Sub( REAL* x,    char* ch_ptr, REAL* y,
     int   x_size, int ch_len, int   y_size )
{
  ...
}
```

Others

The bulk of Fortran compilers add the length as `int` only for type `CHARACTER`:

```
void Sub( REAL* x, char* ch_ptr, REAL* y,
                     int ch_len )
{
  ...
}
```

Apollo

The only remaining exception is the Apollo `ftn` compiler which passes the length in a `short*`:

```
void Sub( REAL* x, char* ch_ptr, REAL* y,
                  short* ch_ref )
{
  int ch_len = *ch_ref;
  ...
}
```

KUIP offers a number of convenience functions (`fstrdup`, `fstrset`, ...) to convert between fixed length, blank-filled Fortran `CHARACTER` buffers and null-terminated, blank-trimmed C strings. For details refer to the KUIP manual.

4- Calling Fortran from C

Calling a Fortran routine from C is basically the reverse operation of providing a C routine which is Fortran callable. One has to know the correct spelling for the external identifier, for numeric variables pass the address instead of the value itself, and for `CHARACTER` items construct the appropriate argument list.

We continue to use the example of a subroutine with the same calling sequence `SUB(X,CH,Y)`. In the following code fragments we assume that

```
  REAL  x, y;
  char* ch_ptr;
  int   ch_len;
```

are already filled and that we want to pass their values to Fortran. Again the platform dependence in passing character data causes the main difficulty.

VMS

On VMS we have to fill the elements of the string descriptor and then pass its address:

```
#include <descrip.h>

  struct dsc$descriptor_s ch_dsc;
  ch_dsc.dsc$b_dtype   = DSC$K_DTYPE_T;
  ch_dsc.dsc$b_class   = DSC$K_CLASS_S;
  ch_dsc.dsc$w_length  = ch_len;
  ch_dsc.dsc$a_pointer = ch_ptr;
  Sub( &x, &ch_dsc, &y );
```

Cray

On Cray a predefined macro packs the address and length information into a `cft77` descriptor:

```
#include <fortran.h>

  Sub( &x, _cptofcd(ch_ptr,ch_len), &y );
```

NeXT

```
  Sub(        &x,  ch_ptr,        &y,
      (sizeof x), ch_len, (sizeof y) );
```

Apollo ftn

The byte ordering of the Motorola-68k CPU does not allow the passing of the address of a 4-byte `int` if a 2-byte `short` is expected. Therefore for the Apollo `ftn` compiler we have to use an auxiliary `short`:

```
  short ch_ref = ch_len;
  Sub( &x, ch_ptr, &y,
          &ch_ref );
```

Others

For all other compilers except IBM-C/370 the call is straightforward:

```
  Sub( &x, ch_ptr, &y,
           ch_len );
```

C/370

The IBM C/370 compiler **does not** allow to call a VS-Fortran routine with `CHARACTER` arguments. The naive approach

```
#pragma linkage(SUB,FORTRAN)

  int x_size = (sizeof x);
  int y_size = (sizeof y);
  Sub( &x,       ch_ptr, &y,
       &x_size, &ch_len, &y_size );
```

does not work. VS-Fortran passes the argument count implicitly by setting the sign bit of the last word in the argument block. uses argument blocks containing the reference addresses.

For routines with `CHARACTER` arguments both the words at positions N and 2N must be flagged. The `#pragma linkage` instructs the C compiler to create a Fortran-style argument block with the last word flagged. Unfortunately there is no way to set the sign bit also in the middle of the argument block. Attempts to patch the argument list such as

```
  REAL* y_ref =
    (REAL*)(0x80000000 | (int)&y);
  Sub( &x,       ch_ptr,  y_ref,
       &x_size, &ch_len, &y_size );
```

do not succeed. (It seems that C/370 clears the sign bit again before storing the address into the argument block.)

The work-around is to use a wrapper routine which receives the character arguments as Hollerith data and copies them into temporary `CHARACTER` variables. For example,

```
#pragma linkage(K77XCX,FORTRAN)
#pragma linkage(SUB,FORTRAN)
extern SUBROUTINE SUB;

  SUBROUTINE* sub_ptr = SUB;
  K77XCX( &sub_ptr, &x, ch_ptr, &ch_len, &y );
```

with

```
      SUBROUTINE K77XCX(SUB,X1,K2,L2,X3)
      EXTERNAL SUB
      PARAMETER(MBIG=256)
      CHARACTER*(MBIG) C2
      CALL UHTOC(K2,4,C2,L2)
      CALL SUB(X1,C2(:L2),X3)
      CALL FMEMCPY(K2,C2(:L2))
      END
```

Obviously, `MBIG` should be set to the maximum string length ever expected. `FMEMCPY` is a Fortran callable version of `memcpy` provided by KUIP. The `CHARACTER` data has to be copied back into the C string in case that it is an output parameter. (`UCTOH` cannot be used because there the number of bytes copied is rounded up to the next multiple of 4.)

The disadvantage is that each type of argument list requires its own wrapper. This is not foreseen in the `cfortran.h` approach and is the reason why it cannot be used for C/370.

5- Input/Output

With few exceptions, mixing Fortran and C input/output calls is asking for trouble. Often the runtime libraries of both languages implement separate buffering schemes in order to improve performance.

Terminal Input

One should never mix `READ(5,...)` and `fgets(...,stdin)` requests. On most systems it works as long as the input device is really a terminal. However, at the latest when `stdin` is connected to a file or a pipe the Fortran `READ` will suck in more than just its share to satisfy the immediate request. Any following `fgets` will then access the wrong information (or report end-of-file).

Terminal Output

Mixing `WRITE(6,...)` and `fprintf(stdout,...)` is safe as long as the output device is the terminal. When `stdout` is redirected the C stream functions usually start buffering the output.

On a number of Unix systems (RS/6000, Alpha/OSF, Ultrix) the Fortran runtime library does not use the C stream functions, and the Fortran and C output buffers are flushed out of sequence. The C runtime libraries on these systems can be told to limit themselves to line buffering by

```
  if( !isatty( fileno(stdout) ) )
    setlinebuf( stdout );
```

It seems that the Fortran runtime libraries are already disciplined enough to use for `WRITE(6,...)` and `PRINT ...` only line buffering. Note that `setlinebuf` is not a standard Unix function. Hopefully, if it is not contained in `stdio.h`, it also is not necessary.

Formatted I/O

There is no general way to associate a Fortran logical unit number with a C `FILE*` stream. KUIP offers C functions `ku_read` and `ku_write` as a convenient interface to Fortran I/O statements for reading/writing complete lines. For details refer to the KUIP manual.

(Note that even in Fortran a `READ(LUN,'(A)') CHLINE` is not portable because the VS-Fortran runtime library is not able to handle V-format files.)

Unformatted I/O

Some systems provide a function `getfd` to inquire about the C file descriptor associated to a Fortran logical unit number. This is however non-standard, and often Fortran unformatted sequential I/O adds system-dependent control information to allow for variable length records.

The CIO package in KERNLIB provides a Fortran callable set of interface routines to the C `read` and `write` functions in order to perform system-independent unformatted I/O, both for sequential and direct access with fixed record sizes.

6- Initialization

On many systems the runtime libraries for a given language has to be initialized — a task which is then performed by the main program.

6.1- Fortran main program

With a Fortran main program calling C functions directly or indirectly the following points have to be respected.

C/370

The C/370 runtime library is automatically initialized when the first C routine is called. The system resources allocated there are released again as soon as the Fortran routine, from which the C routine triggering the initialization was called, returns. Calling another C routine after that does not cause a re-initialization and the program usually crashes because the `malloc` heap has become invalid.

This effect can be safely avoided by calling the first C routine immediately in the Fortran main program. KERNLIB contains a dummy routine `INITC` for this purpose.

When executing a module containing C code the C runtime libraries have to be included in the `GLOBAL` library lists. The `CERNLIB` command is the most convenient call to this. Note that the C compiler itself is a C program and will abort with an unintelligible message if the libraries are not defined.

VMS

A Fortran program should call `VAXC$CRTL_INIT` (either from Fortran or from C) before calling any functions from the C runtime library. There can be strange effects if this rule is not obeyed.

6.2- C main program

Our experience with a C main program calling Fortran routines is limited. One known restriction is that the `GETARG` routine for retrieving the command line arguments in Fortran cannot be used. Calling this routine usually results in unresolved externals called `f77_argc` or similar.